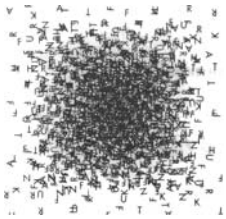
Fachbereich Informatik und Ingenieurwissenschaften der
Fachhochschule Frankfurt Main – University of Applied Sciences
Prof. Dr. Peter Nauth
Koreferent: Prof. Dr. Andreas Pech



# Masterthesis

## Simulation von bewegten Objekten in einem closed-loop Visionsystem

von

**Dipl.-Ing. Mirko Radowitz**

Die Arbeit wurde erstellt in Kooperation mit dem

Honda Research Institute Europe GmbH

Betreuer:
Dipl.-Ing. Mark Dunn
Dr.-Ing. Michael Gienger

**September 2006**

## Erklärung

„Ich versichere, dass ich meine Masterarbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen."

Frankfurt, 5. Oktober 2006

_____

Mirko Radowitz

# Danksagung

Ich möchte mich an dieser Stelle ganz besonders bei meinen Betreuern Herrn Dipl.-Ing. Mark Dunn und Herrn Dr.-Ing. Michael Gienger für die sehr gute Betreuung und Hilfestellung während der Arbeit bedanken. Ihre fachlich kompetente und menschlich nette Betreuung war mir eine große Hilfe.

Auch bedanken möchte ich mich bei Herrn Dr.-Ing. Christian Goerick für die Ermöglichung dieser Masterarbeit und bei allen anderen Kollegen vom Honda Research Institute.

Weiterer Dank geht an meinen betreuenden Dozenten von der Fachhochschule Frankfurt am Main Herrn Prof. Dr. Peter Nauth.

Auf diesem Wege möchte ich mich auch bei Frau Dipl.-Kff. Swenja Kaschel für Ihre Unterstützung bedanken.

Mirko Radowitz

# Inhaltsverzeichnis

Erklärung ..... 1-III

Inhaltsverzeichnis ..... V

Abbildungsverzeichnis ..... VII

Abkürzungsverzeichnis ..... IX

1 Einleitung ..... 10

1.1 Das Honda Research Institute Europe ..... 10
1.2 Motivation ..... 10
1.3 Aufgabenstellung und Zielsetzung ..... 11
1.4 Aufbau der Arbeit ..... 13

2 Theoretische Grundlagen ..... 14

2.1 Definitionen ..... 14
2.1.1 Simulation ..... 14
2.1.2 Simulationsprogramm ..... 14
2.1.3 Simulator ..... 14
2.1.4 Regelung (Regelkreis) ..... 14
2.2 Mechanik starrer Köper ..... 15
2.2.1 Der starre Körper ..... 15
2.2.2 Kinematik des starren Körpers ..... 16
2.2.3 Dynamik: Kräfte und Drehmomente ..... 19
2.3 Kollisionserkennung ..... 20
2.3.1 Hüllkörper ..... 21
2.3.2 Hüllkörperhierarchien ..... 23
2.3.3 Raumaufteilungen ..... 24
2.3.4 Basiskollisionserkennung ..... 25
2.4 Physik-Engines ..... 25
2.5 Open Dynamics Engine ..... 26
2.5.1 Struktur von ODE ..... 27
2.5.2 Die Welt ..... 29
2.5.3 Der Körper ..... 30
2.5.4 Das Gelenk ..... 31

2.5.5 Gelenkfehler ... 33
2.5.6 Weiche Bedingungen ... 34
2.5.7 Einsatz von CFM und ERP ... 34
2.5.8 Reibung ... 35
2.5.9 Kollisionsräume ... 35
2.5.10 Die Geometrie-Objekte ... 35
2.5.11 Kollisionserkennung ... 36
2.5.12 Simulation / Integration ... 39
2.5.13 Simulationsprogramm ... 41
2.6 ViToSA ... 41

3 Entwicklung eines Software-Simulationssystems ... 44

3.1 Projektplanung und Vorbereitung ... 44
3.2 Anforderungen ... 45
3.2.1 Anforderungen an das Simulationssystem ... 45
3.2.2 Anforderung an den Simulator ... 46
3.2.3 Anforderung an die Manipulation ... 47
3.2.4 Anforderung an die Visualisierung ... 47
3.3 Simulationssystem ... 48
3.3.1 Konzeption des Simulationssystems ... 48
3.3.2 Schnittstellen Design ... 49
3.3.3 Ablauf der Kommunikation ... 52
3.4 Simulator ... 53
3.4.1 Konzept ... 53
3.4.2 Implementierung ... 54
3.4.3 Simulationsablauf ... 56
3.4.4 Testszenarios ... 59
3.5 Manipulation ... 59
3.6 Visualisierung ... 60
3.7 Testszenarios ... 62

4 Zusammenfassung und Ausblick ... 64

Literaturverzeichnis ... LXVIII

Anhang ... LXIX

# Abbildungsverzeichnis

Abbildung 1-1: Darstellung des ersten Regelkreises....12
Abbildung 1-2: Darstellung des zweiten Regelkreises....12
Abbildung 2-1: Starrer Körper mit lokalem Koordinatensystem....16
Abbildung 2-2: Darstellung einer Translationsbewegung....17
Abbildung 2-3: Darstellung einer Rotationsbewegung....18
Abbildung 2-4: Darstellung einer allgemeinen Bewegung....19
Abbildung 2-5: Kugel als Hüllkörper....21
Abbildung 2-6: Achsenorientierter Quader als Hüllkörper....22
Abbildung 2-7: Beliebig orientierter Quader....23
Abbildung 2-8: Simulation von zwei miteinander verbundenen Körpern....27
Abbildung 2-9: Schematische Darstellung der Objektstruktur in ODE....27
Abbildung 2-10: Verbindung von Körpern mit Gelenken....28
Abbildung 2-11: Struktur einer virtuellen Welt in ODE....29
Abbildung 2-12: Kugel-Gelenk....31
Abbildung 2-13: Scharnier-Gelenk....31
Abbildung 2-14: Schiebe-Gelenk....32
Abbildung 2-15: Universal-, Schanier-Typ2- und Kontakt-Gelenk....32
Abbildung 2-16: Funktionsweise des Error Reduction Parameters....34
Abbildung 2-17: Die verschiedenen Geometrien in ODE....36
Abbildung 2-18: Körper in ODE mit mehreren Kollisionsobjekten....36
Abbildung 2-19: Aufbau und Ablauf einer Simulation in ODE....40
Abbildung 2-20: Screenshot von ViToSA....42
Abbildung 2-21: Konzeption von ViToSA....43
Abbildung 3-1: Projektplan....45
Abbildung 3-2: Konzeption des Simulationssystems....48
Abbildung 3-3: Datenstruktur SharedDateVisualisation....51

Abbildung 3-4: Datenstruktur SharedDataManipulation .....52
Abbildung 3-5: Ablauf der Kommunikation .....53
Abbildung 3-6: Architektur des Simulators.....54
Abbildung 3-7: Simulationsbibliothek HSimL .....55
Abbildung 3-8: UML-Diagramm der HSimL-Bibliothek .....56
Abbildung 3-9: Programmablaufplan des Simulators .....56
Abbildung 3-10: Programmablauf der Initialisierungsphase .....57
Abbildung 3-11: Programmablauf der Simulationsschleife .....58
Abbildung 3-12: Programmablauf des Einleseprozesses .....60
Abbildung 3-13: Kommunikationsablauf zwischen SHM und ViToSA .....61
Abbildung 3-14: Screenshot einer Beispiel-Simulationsumgebung mit ViToSA .....62
Abbildung 3-15:Darstellung des Test-Regelkreises (Closed-Loop) .....63

# Abkürzungsverzeichnis

| | |
|---|---|
| 3D | Dreidimensional |
| AABB | Axis-Aligned Bounding Box |
| API | Application Programming Interface |
| ASIMO | Advanced Step in Innovative Mobility |
| BSD | Berkeley Software Distribution |
| BSP | Binary Space Partitioning |
| DDD | Data Display Debugger |
| GCC | GNU Compiler Collection |
| gcc | GNU C Compiler |
| GNU | GNU's Not Unix |
| GPL | General Public License |
| HRI | Honda Research Institute |
| HSimL | HRI Simulation Library |
| ID | Identifikationsnummer |
| IPC | Interprozesskommunikation |
| K-DOP | K - Discrete Oriented Polytop |
| MUTEX | Mutual Exclusion |
| OBB | Oriented Bounding Box |
| ODE | Open Dynamics Engine |
| OPCODE | Optimized Collision Detection |
| OpenGL | Open Graphics Library |
| POSIX | Portable Operating System Interface for UniX |
| Qt | Quasar toolkit |
| RTBOS | Real Time Brain Operating System |
| SHM | Shared Memory |
| SVN | Subversion |
| UI | User Interface |
| ViToSA | Visualisation Tool for Simulation Applications |

# 1 Einleitung

## 1.1 Das Honda Research Institute Europe

Das Honda Research Institute Europe GmbH (HRI) ist ein Tochterunternehmen des japanischen Automobilherstellers Honda Motor Co. Ltd. Es wurde zusammen mit den Schwesterunternehmen in Japan und den USA im Januar 2003 gegründet und ist im Bereich der Grundlagenforschung für Intelligente Systeme tätig.

Ein wesentliches Ziel von HRI ist es die Mechanismen und die Entwicklung von Informationsprozessen in biologischen Systemen zu untersuchen, die grundlegenden Prinzipien zu identifizieren und sie anschließend in technischen Systemen einzufügen.

HRI arbeitet mit mehreren nationalen und internationalen Forschungsinstituten und Universitäten zusammen und verknüpft somit die universitäre Wissenschaft mit der praktischen Forschung in verschiedenen Disziplinen.[1]

## 1.2 Motivation

Die Entwicklung eines humanoiden Roboters sowie das erstellen und testen eines komplexen Visionsystems ist eine sehr komplizierte und anspruchsvolle Aufgabe. Dies wird zusätzlich durch den Mangel an Sensoren und Aktoren erschwert, da sie nur begrenzt zur Verfügung stehen. Ferner sind viele aufwendige Testverfahren notwendig, um eine fehlerfreie Funktion der Hard- und Software zu gewährleisten. Gerade aber bei den Testdurchläufen wird die empfindliche und teure Hardware sehr belastet und es besteht die Gefahr einer Beschädigung. Dazu kommt der Aufbau von komplizierten Testumgebungen, die nötig sind, um das entwickelte Verhalten zu überprüfen und zu testen.

Ein weiteres Problem ergibt sich bei der Nutzung der Hardware von mehreren Wissenschaftlern, da sie immer nur für einen Versuch genutzt werden kann und es somit automatisch zu Engpässen kommt.

Die Entwicklung einer Software-Simulationsumgebung für den Roboter gibt die Möglichkeit diesen Problemen entgegenzuwirken und den Entwicklungsprozess zu beschleunigen.

[1] Siehe Honda Research Institute Europe GmbH, www.honda-ri.de (2006)

## 1.3 Aufgabenstellung und Zielsetzung

Es ist ein Software-Simulationssystem zu entwickeln, welches ermöglicht die Dynamik und Kinematik von Objekten (starren Körpern) zu simulieren, visualisieren und zu manipulieren.

Für die Simulation der Dynamik und Kinematik starrer Körper ist eine Bibliothek zu konzipieren und zu implementieren, mit deren Hilfe es möglich wird, eine virtuelle Welt zu erstellen. In dieser virtuellen Welt ist die Umgebung des Roboters nachzubilden. Die Bibliothek soll flexibel und erweiterbar sein. Die Basis dafür ist die frei erhältliche ODE Bibliothek (Open Dynamics Engine).

Die virtuelle Welt soll über eine externe (Manipulations-) Schnittstelle konfigurierbar bzw. manipulierbar sein. Das heißt es muss mögliche sein über ein entsprechendes Eingabegerät, wie z. B. eine (Space-) Maus oder einfach über die Tastatur mit der virtuellen Welt zu interagieren, indem Objektpositionen und andere Objekteigenschaften verändert werden können.

Die Visualisierung der virtuellen Welt soll mit dem vom HRI entwickelten Grafikprogramm ViToSA (Visualisation Tool for Simulation Applications) erfolgen. Um dies zu ermöglichen ist ein Plug-In zu konzipieren und zu implementieren.

Für das Testen des Simulationssystems sind zwei unterschiedliche Regelkreise (Closed-Loops*)* mit dem vom HRI verwendeten Komponenten-/Integrationsmodell vorgesehen. Der erste einfachere Regelkreis (siehe Abbildung 1-1) besteht aus den zwei Komponenten **Simulator** und **Robot-Control**. Die bereits bestehende Komponente Robot-Control simuliert den Roboter. Die Simulator Komponente ist im Rahmen dieser Arbeit zu implementieren und soll die virtuelle Welt, also die Umgebung des Roboters simulieren. Sie generiert symbolische Informationen von der virtuellen Umgebung und übergibt diese an die Robot-Control Komponente. Symbolische Informationen können z. B. die Koordinaten des Massenschwerpunktes eines starren Körpers sein. Die Robot-Control Komponente kann auf Basis dieser Daten wiederum Änderungen in der virtuellen Welt vornehmen. So könnte z. B. die Simulator Komponente ein Tisch, auf dem sich mehrere Objekte befinden, simulieren und die Positionen des Tisches und der Objekte an die Robot-Control Komponente übergeben. Diese kann nun in die Simulation eingreifen und die Positionen der Objekte in der virtuellen Welt verändern wie z. B. ein Objekt greifen und neben dem Tisch wieder loslassen. Das Objekt würde dann aufgrund der Schwerkraft auf den Boden herunter fallen.

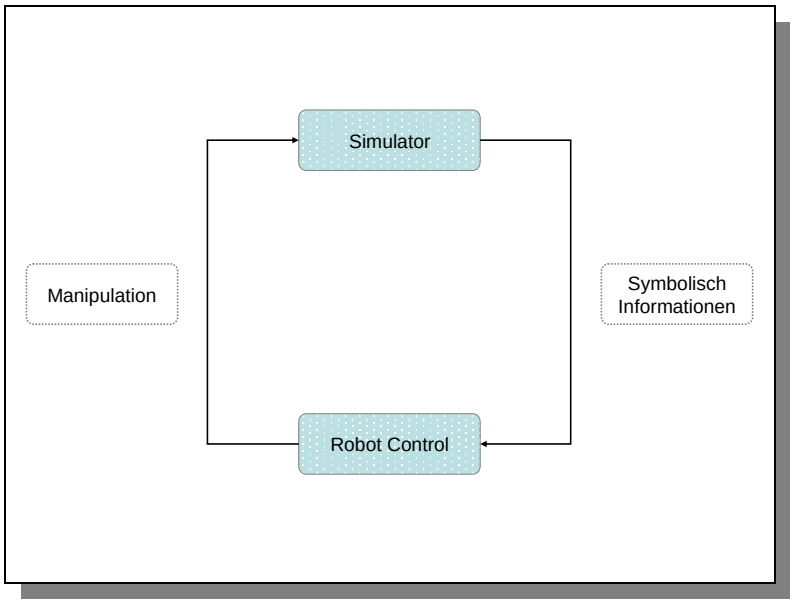



Abbildung 1-1: Darstellung des ersten Regelkreises, Quelle: Eigene Darstellung.

Der zweite Regelkreis (siehe Abbildung 1-2) basiert auf dem ersten Regelkreis jedoch mit dem Unterschied, dass die symbolischen Informationen zunächst visualisiert werden. Aus den visuellen Daten werden anschließend Bild-Sequenzen generiert und einem Vision-system übergeben. Das Visionsystem extrahiert aus den visuellen Informationen wiederum Kommandos zur Steuerung des Roboters, die der Robot-Control Komponente übergeben werden. Diese kann nun wieder aufgrund der Informationen in die virtuelle Welt eingreifen und sie manipulieren.

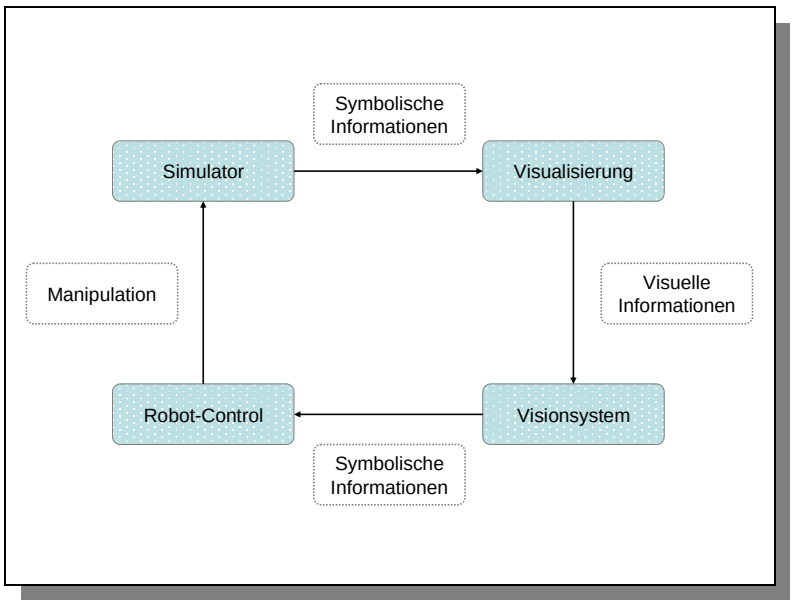



Abbildung 1-2: Darstellung des zweiten Regelkreises, Quelle: Eigene Darstellung.

Das Ziel der Arbeit ist es mit dem erstellten Software-Simulationssystem eine virtuelle Umgebung für den simulierten Roboter zu erzeugen und diese beiden Komponenten miteinander zu koppeln.

## 1.4 Aufbau der Arbeit

Das 2. Kapitel geht auf die theoretischen Grundlagen der in dieser Arbeit angewandten Konzepte ein. Zunächst werden wesentliche Definitionen aufgeführt, die im Zusammenhang mit dieser Arbeit stehen. Danach wird die Mechanik starrer Körper erläutert und die gängigsten Verfahren zur Kollisionserkennung beschrieben. Der nächste Abschnitt stellt anschließend die zur Verfügung stehenden Physik-Engines vor und beschreibt die letztendlich eingesetzte ODE-Bibliothek zur Simulation der Dynamik. Der letzte Abschnitt dieses Kapitels befasst sich mit dem Visualisierungsprogramm ViToSA.

In Kapitel 3 wird schließlich die Umsetzung der Masterarbeit beschrieben. Nach der Planung des Projekts folgen die Anforderungen an das Gesamtsystem bzw. an die einzelnen Komponenten. Daraufhin wird das Simulationssystem als Ganzes beschrieben und im Anschluss daran die einzelnen Module Simulator, Benutzerschnittstelle und Visualisierung genauer betrachtet. Am Ende des Kapitels folgt die Beschreibung des Testszenarios.

Im abschließenden 4. Kapitel werden die Ergebnisse zusammengefasst, entstandene Probleme besprochen und ein Ausblick für die weitere Entwicklung gegeben.

# 2 Theoretische Grundlagen

## 2.1 Definitionen

Dieses Kapitel soll grundlegende Begriffe kurz beschreiben und somit eine Basis für die darauf folgenden Kapitel geben.

### 2.1.1 Simulation

Simulation ist die Bezeichnung für die Nachbildung und Analyse eines Systems oder Prozesses durch ein vereinfachtes Modell. Durch eine Simulation kann auf einfache und zeitsparende Weise das Verhalten von Objekten nachgebildet werden, anstatt komplizierte Verfahren zur Untersuchung anzuwenden.[2]

Für die Simulation von bewegten Objekten wird das vereinfachte Modell von starren Körpern angewendet, die in der Realität so nicht vorkommen. Dies vereinfacht die Simulation erheblich.

### 2.1.2 Simulationsprogramm

Ein Simulationsprogramm bildet das Verhalten eines Systems oder Prozesses durch ein Modell auf einem Rechner ab. Die Verhaltensregeln sind durch mathematische Gleichungen beschrieben und mit Hilfe des Computers werden alle Lösungsmöglichkeiten durchgespielt.

### 2.1.3 Simulator

Ein Simulator ist ein Gerät, System oder Computer mit dem bestimmte Eigenschaften eines Systems bzw. Prozesses nachgebildet werden können.

### 2.1.4 Regelung (Regelkreis)

Die Regelung ist ein Vorgang bei dem eine bestimmte Größe X kontinuierlich erfasst und mit einem vorgegebenen Sollwert abgeglichen wird. Im Gegensatz zur Steuerung ist die Regelung ein geschlossener Wirkungskreis.[3]

---

[2] Vgl. hierzu und im Folgenden Myers Enzyklopädisches Lexikon (1977), Band 21, S. 746.
[3] Vgl. Myers Enzyklopädisches Lexikon (1977), Band 19, S. 710.

## 2.2 Mechanik starrer Köper

Dieses Kapitel stellt die Grundlagen für die Bewegung von Körpern (Objekten) vor, die als Basis für die nachfolgenden Kapitel dienen.

Die klassische Mechanik beschäftigt sich mit der Beschreibung und Berechnung der Bewegung von Teilchen (Massepunkten) und Körpern. Es geht also um das Aufstellen und Lösen von Bewegungsgleichungen. Die Mechanik kann in Kinematik und Dynamik unterteilt werden. Die Kinematik beschreibt die Bewegung von Körpern ohne die Berücksichtigung der einwirkenden Kräfte, also nur den zeitlichen Verlauf. Die Bewegung wird durch die Größen Ort/Weg, Geschwindigkeit und Beschleunigung beschrieben. Die Dynamik beschäftigt sich im Gegensatz zur Kinematik mit der durch Kräfte hervorgerufenen Bewegung, also mit der Ursache und Wirkung der Bewegung. Sie beschreibt somit die Änderung der Bewegungsgrößen Weg, Geschwindigkeit und Beschleunigung unter der Einwirkung von Kräften und Momenten.[4]

### 2.2.1 Der starre Körper

Honerkamp und Römer beschreiben in ihrem Werk „Klassische Theoretische Physik“ einen starren Körper wie folgt: „Ein Körper wird als starr bezeichnet, wenn er als unverformbar angesehen werden kann, d.h. wenn in guter Näherung die Abstände zwischen allen seinen Teilen unverändert bleiben.“[5] D. h. ein starrer Körper ist ein System von Massenpunkten mit konstanten Abständen zueinander. In der Realität ist ein Körper sicherlich nie völlig starr, sondern unterliegt Veränderungen und Deformationen, wenn er bspw. rotiert. Deformationen wiederum verteilen die Masse eines Körpers neu und würden die Bewegung nur noch weiter verkomplizieren. Daher wird der starre Körper als ein Ersatz für die Realität benutzt, der sozusagen nur ein idealisiertes Modell der theoretischen Physik ist.

Ein starrer Körper besitzt 6 Freiheitsgrade. Diese geben an, wie viele von einander unabhängige Bewegungen ein starrer Körper durchführen kann. Dazu gehören die drei Freiheitsgrade der Translation, also die Bewegung in den drei räumlichen Dimensionen Länge, Breite und Höhe, sowie die drei Freiheitsgrade der Rotation, also die Drehung um die x-, y- und z-Achse. Die Lage des Körpers ist festgelegt durch Angabe von drei Punkten. Um die Hand-

[4] Vgl. Gross/Hauger/Schnell (1998), S. 1 f.
[5] Vgl. Honerkamp/Römer (1989), S. 81.

habung der Bewegungen eines starren Körpers zu vereinfachen, wird die Translations- und Rotationsbewegung jeweils einzeln behandelt.

Für die Bewegung eines starren Körpers ist ein geeignetes Koordinatensystem nötig. Dieses ist fest und kann weder verschoben noch rotiert werden (Inertialsystem). Es symbolisiert somit den Welt-Raum. Um die Bewegungen eines Körpers einfacher zu beschreiben wird ein zusätzliches körperfestes Koordinatensystem mit dem Körper assoziiert, wie in der Abbildung 2-1 grafisch dargestellt. Dieses Koordinatensystem hat seinen Ursprung im Zentrum des Körpers $R_M$ (Schwerpunkt).

$$R_M = \frac{\sum m_i r_i}{\sum m_i} = \frac{\sum m_i r_i}{M}$$ (Ort des Massenmittelpunktes)

$m_i =$ Massenelement , $r_i =$ Radius des Massenelements

Dieser körperbezogene fixe Schwerpunkt $R_M$ vereinfacht den Umgang mit der Bewegung eines starren Körpers.

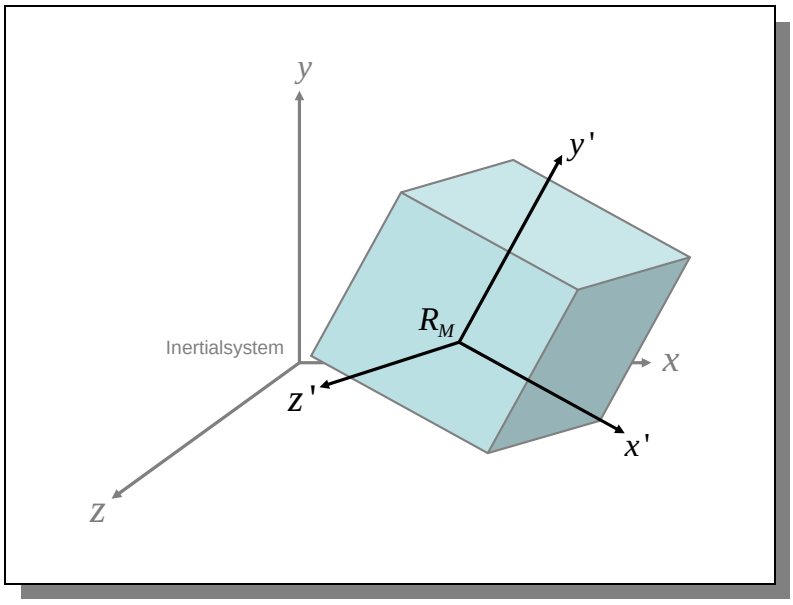



Abbildung 2-1: Starrer Körper mit lokalem Koordinatensystem, Quelle: Eigene Darstellung.

### 2.2.2 Kinematik des starren Körpers

**Translationsbewegung**

„Translation nennt man eine Bewegung, bei der die Verbindungsstrecke zwischen zwei beliebigen Punkten A und B eines Körpers ihre Richtung nicht ändert […]. Alle Punkte erfahren

dann in der Zeit $dt$ die gleiche Verschiebung $dr$. Damit sind auch die Geschwindigkeiten $v$ und die Beschleunigungen $a$ für alle Punkte des Körpers gleich:

$$v = \frac{dr}{dt},\ a = \frac{dv}{dt} = \frac{d^2r}{dt^2}$$

Die Bahnkurven, die von verschiedenen Körperpunkten durchlaufen werden, haben alle die gleiche Form. Bei der Translation ist demnach die Bewegung eines beliebigen Körperpunktes repräsentativ für die Bewegung des ganzen Körpers."[6] Die Abbildung 2-2 macht die Translation anhand einer Grafischen Darstellung deutlich.

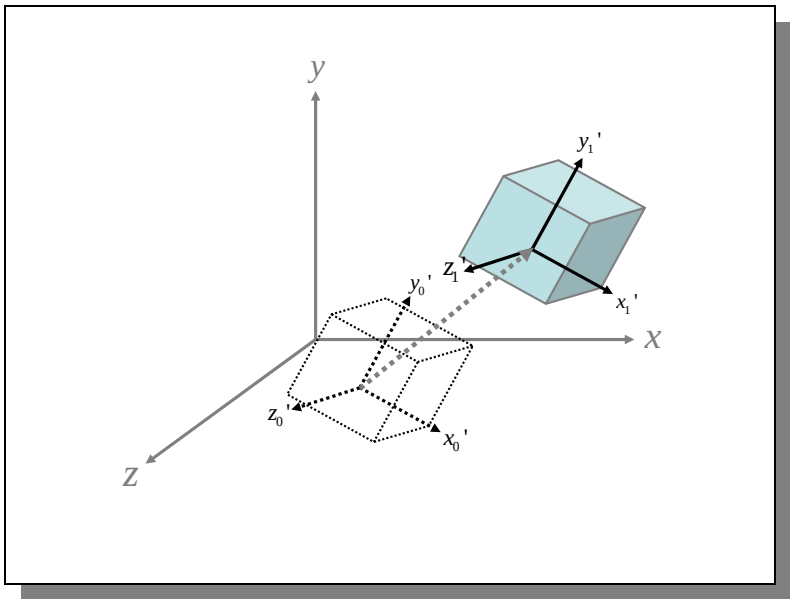



Abbildung 2-2: Darstellung einer Translationsbewegung. Quelle: Eigene Darstellung.

**Rotationsbewegung**

„Bei einer Rotation bewegen sich alle Punkte des Körpers um eine gemeinsame Drehachse. Ist die Lage dieser Achse im Raum unveränderlich, so spricht man von einer Rotation um eine feste Achse. Geht die Drehachse dagegen nur durch einen raumfesten Punkt (Fixpunkt) und verändert ihre Richtung mit der Zeit, so bezeichnet man dies als eine Rotation um einen Fixpunkt.

Wir betrachten zunächst die Rotation eines Körpers um eine feste Achse […]. In diesem Fall bewegen sich die Punkte auf Kreisbahnen, deren Ebenen jeweils senkrecht zur Drehachse

[6] Vgl. Gross/Hauger/Schnell (1998), S. 103.

stehen. Die Fahrstrahlen zu allen Körperpunkten überstreichen in gleichen Zeiten den gleichen Drehwinkel $\varphi$. Demnach sind die Winkelgeschwindigkeit $\omega$ und die Winkelbeschleunigung $\alpha$ für alle Punke gleich.

$$\omega = \frac{d\varphi}{dt},\ \alpha = \frac{d\omega}{dt} = \frac{d^2\varphi}{dt^2},$$

$d\varphi =$ Änderung des Drehwinkels, $dt$ = Änderung der Zeit

Wir wenden uns nun der Rotation um einen raumfesten Punkt A zu […]. Die momentane Lage der Drehachse sei durch den Einheitsvektor $e_w$ gekennzeichnet. Führt der Körper in der Zeit $dt$ eine Drehung mit dem Drehwinkel $d\varphi$ um die augenblickliche Drehachse aus, so bewegen sich alle Körperpunkte momentan auf Kreisbahnen.“[7]

Die Rotation um einen raumfesten Punkt A kann hinsichtlich des Ergebnisses stets auch durch die Rotation um eine durch den festen Punkt gehende Achse erzielt werden (Eulersches Theorem).Die Abbildung 2-3 verdeutlicht die Rotation in einer grafischen Darstellung.

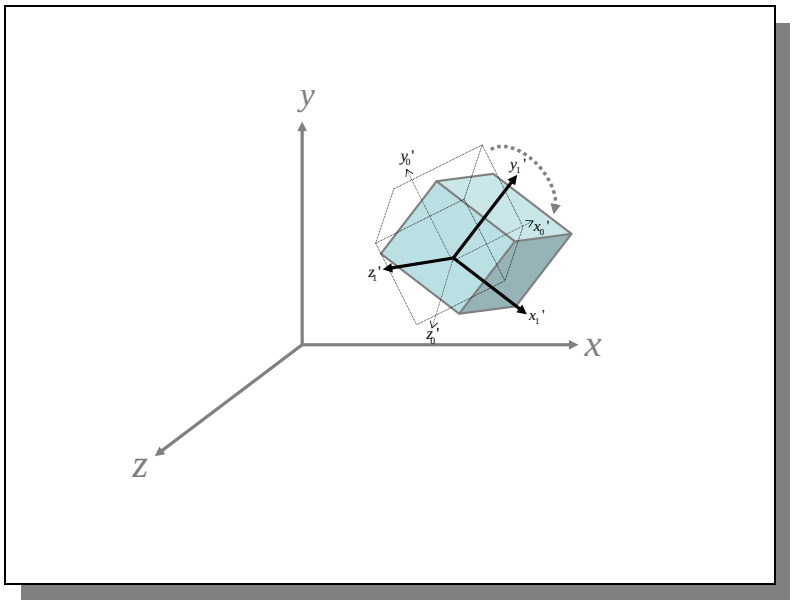



Abbildung 2-3: Darstellung einer Rotationsbewegung, Quelle: Eigene Darstellung.

[7] Vgl. Gross/Hauger/Schnell (1998), S. 104 ff.

### Allgemeine Bewegung

Die allgemeine Bewegung eines starren Körpers setzt sich aus einer Translation und einer Drehung des Gesamtsystems zusammen (Theorem von Chasles).[8]

Es reicht aus drei Punkte innerhalb eines starren Körpers zu betrachten, um dieses Theorem zu verdeutlichen. Alle drei Punkte sind von einer Startposition in eine Endposition überführbar, indem sie parallel verschoben und anschließend um die entsprechende Achse gedreht werden. Die Abbildung 2-4 stellt die allgemeine Bewegung grafisch dar.

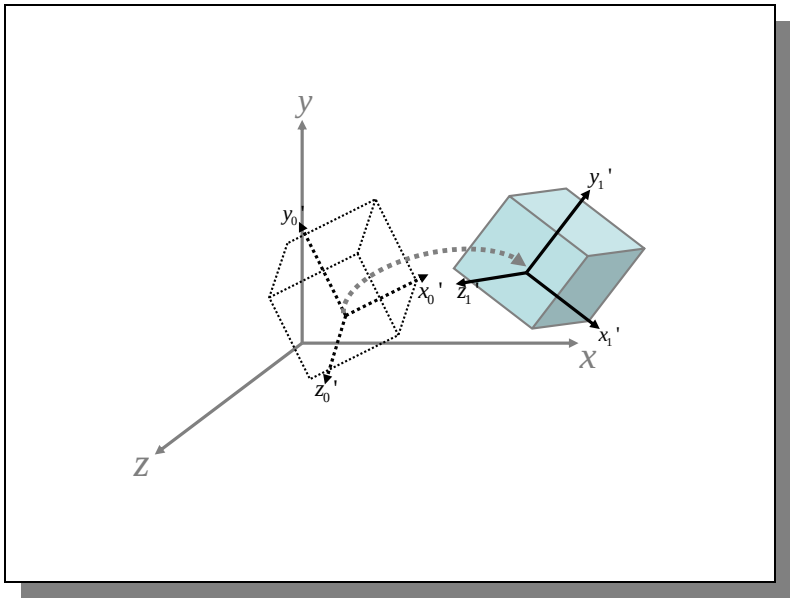



Abbildung 2-4: Darstellung einer allgemeinen Bewegung, Quelle: Eigene Darstellung.

### 2.2.3 Dynamik: Kräfte und Drehmomente

Das erste Bewegungsgesetz von Newton besagt, dass ein Körper sich in Ruhe befindet oder seine Geschwindigkeit konstant ist, solange keine äußere Kraft wirkt. Demzufolge setzt sich ein Körper in Bewegung oder in Drehbewegung, wenn an ihm eine Kraft oder ein Drehmoment wirkt. Das Drehmoment M spielt für die Rotation die gleiche Rolle wie die Kraft für die Translation.[9]

Die Translationsbewegung eines starren Körpers kann durch den Impuls charakterisiert werden. Für Rotationsbewegungen ist eine weitere physikalische Größe, der so genannte Dreh-

[8] Vgl. hierzu und im Folgenden Dreizler/Lüdde (2003), S. 316.
[9] Vgl. hierzu und im Folgenden Baraff (1997), S. 13 ff.

impuls zu definieren. Er charakterisiert die Drehbewegung in ähnlicher Weise wie der Impuls die Translationsbewegung.

Der Drehimpuls $L$ eines um eine feste Achse rotierenden Körpers ist das Produkt aus seinem Trägheitsmoment $I$ (bezüglich dieser Achse) und seiner Winkelgeschwindigkeit $\omega$.

$$L = \mathrm{I}\omega$$

Ähnlich wie die Kraft als zeitliche Änderung des Impulses angesehen werden kann $(F = \frac{dp}{dt})$, gibt es einen Zusammenhang zwischen Drehimpuls und Drehmoment: $(M = \frac{dL}{dt})$.

## 2.3 Kollisionserkennung

Die Kollisionserkennung ist verantwortlich für die Überprüfung, ob sich Objekte in einem Kollisionsraum berühren oder überschneiden. Dadurch soll das gegenseitige Durchdringen von Körpern verhindert werden. Die allgemeine sehr aufwendige Variante um Kollisionen zu erkennen, ist die Brute-Force-Methode bei der jedes einzelne Objekt mit jedem anderen Objekt auf Kollisionen überprüft wird. Dies ist ziemlich zeit- und rechenaufwendig, vor allem weil Kollisionen in der Regel verhältnismäßig selten auftreten.

Um diesen Aufwand zu verkleinern, ist die Kollisionserkennung sehr häufig in zwei Phasen unterteilt. In der ersten Phase wird versucht alle Objektpaare auszuschließen, die nicht kollidieren können, um somit die schnelle Auffindung von Kollisionsbereichen zu ermöglichen. Dafür werden ganz spezielle Algorithmen für eine Raumaufteilung eingesetzt. In der zweiten Phase wird mit Hilfe komplexerer Algorithmen zwischen den übrig gebliebenen Objekten eine Kollisionserkennung durchgeführt.[10]

Zwei spezielle Umstände stellen die Kollisionserkennung vor eine besondere Herausforderung: Eine Große Anzahl an Objekten und geometrisch komplexe Objekte. Hier kann die Anwendung von räumlichen Datenstrukturen die Kollisionserkennung beschleunigen. Zum einen ist es möglich bei einer großen Anzahl an Objekten eine Raumpartitionierung anzuwenden. Zum anderen kann die besonders schwierige Handhabung von geometrisch komplexen Objekten durch Hüllkörper bzw. Hüllkörper-Hierarchien optimiert werden.

Die Kollisionserkennung kann in eine statische und in eine dynamische Kollisionserkennung eingeteilt werden. Die statische Kollisionserkennung entscheidet, ob sich nicht bewegende

[10] Vgl. Eckstein (1998), S. 95.

Objekte überschneiden, während bei der dynamischen Kollisionserkennung überprüft wird, ob sich bewegende Objekte schneiden.[11]

### 2.3.1 Hüllkörper

Die Verwendung von Hüllkörpern vereinfachen komplexe geometrische Strukturen. Dadurch lassen sich die Anwendungen von komplizierten Algorithmen wesentlich beschränken. Hierfür stehen mehrere verschiedene Ansätze, die wiederum verschieden Hüllkörper verwenden, zur Verfügung. Im Folgenden sollen die wichtigsten von ihnen aufgeführt werden.

**Kugel**

Der geometrische Körper wird durch eine Kugel umgeben, wie in der Abbildung 2-5 dargestellt. Kugeln haben den Vorteil, dass der Test auf Kollisionen zwischen zwei Kugeln sehr schnell ist, da lediglich die Abstände ihrer Mittelpunkte berechnet werden müssen. Allerdings haben die meisten Objekte eine orthogonale Struktur, wie z. B. ein Würfel, so dass eine Annäherung an diese Strukturen mit Kugeln bzw. die Hülleffizienz[12] nicht sehr gut ist.[13]

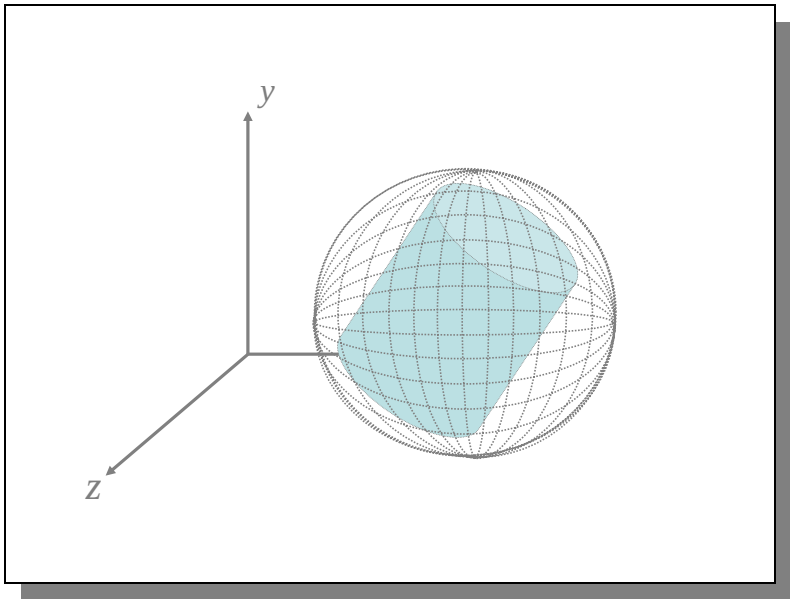



Abbildung 2-5: Kugel als Hüllkörper, Quelle: Eigene Darstellung.

[11] Vgl. Warken (2004), S.5 bzw. Eckstein (1998), S. 19.
[12] Mit Hülleffizienz ist gemeint, wie genau ein Objekt von einer Hülle umgeben wird.
[13] Vgl. hierzu und im Folgenden Eckstein (1998), S. 27 ff.

### Achsenorientierte Quader (AABB)

Bei dem achsenorientierten Quader wird um den geometrischen Körper ein achsenparalleler Quader gelegt, wie in der Abbildung 2-6 zu erkennen ist. Die Quader sind dabei an den Achsen des Koordinatensystems ausgerichtet. Überlappungstests können auf diese Weise sehr schnell ausgeführt werden. Allerdings ist auch hier die Hülleffizienz bei nicht orthogonalen Strukturen oder bei orthogonalen Strukturen, die nicht an den Koordinatenachsen ausgerichtet sind, nicht besonders gut.

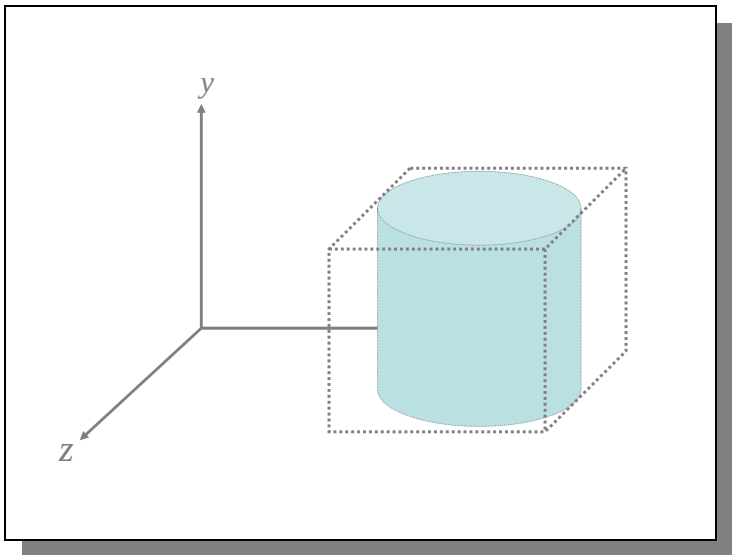



Abbildung 2-6: Achsenorientierter Quader als Hüllkörper, Quelle: Eigene Darstellung.

### Beliebig orientierte Quader (OBB)

Dieses Verfahren funktioniert ähnlich wie der Ansatz des achsenorientierten Quaders. Jedoch werden die Quader nicht an den Achsen des Koordinatensystems ausgerichtet, sondern an den geometrischen Körper selbst (siehe Abbildung 2-7). Dadurch wird die Hülleffizienz stark verstärkt. Allerdings sind die Kollisionstests wesentlich aufwendiger als bei den Kugeln und achsenorientierten Quadern.[14]

[14] Vgl. Eckstein (1998), S. 27 ff.

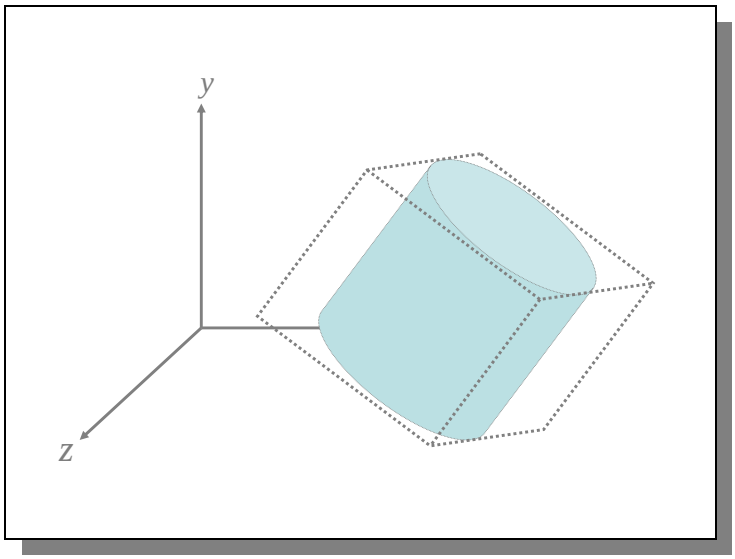



Abbildung 2-7: Beliebig orientierter Quader, Quelle: Eigene Darstellung.

**Konvexe Polyeder beschränkter Richtung (K-DOP)**

Dieses Verfahren ähnelt dem Ansatz der achsenorientierten Quadern, jedoch mit dem Unterschied, dass es mehrere Beschränkungsflächen gibt. Somit hat dieses Verfahren eine wesentlich bessere Hülleffizienz. Bei Bewegungen ist die Berechnung der Endlage allerdings wesentlich aufwendiger.[15]

### 2.3.2 Hüllkörperhierarchien

Durch die Verwendung von Hüllkörpern können Überlappungstests schon wesentlich beschleunigt werden. Eine weitere Möglichkeit die Kollisionserkennung zu verbessern ist die Verwendung von Hüllkörperhierarchien, d. h. ein geometrisches Objekt in einen Baum von immer kleiner werdenden Hüllkörpern aufzugliedern. Dabei wird das gesamte Objekt zunächst durch einen Hüllkörper umgeben. Anschließend wird der Hüllkörper in zwei weitere kleinere Hüllkörper aufgeteilt. Dieser Vorgang kann immer weiter geführt werden. Somit können große Teile schon ziemlich früh von einer Kollision ausgeschlossen werden. Die Überprüfung auf Kollisionen beginnt bei der Wurzel. Wenn eine Kollision erkannt wurde, wird die nächst tiefere Stufe in der Hierarchie betrachtet und nochmals getestet, ansonsten wird die Überprüfung gestoppt.

[15] Vgl. Eckstein (1998), S. 28 ff.

Für den Aufbau der Bäume stehen viele verschieden Varianten zur Verfügung, die entweder eine top-down oder bottom-up Strategie verwenden. Die Bäume bestehen dabei entweder aus Kugel-Hüllkörper, achsenorientierten Quadern, beliebig orientierten Quadern, K-DOPS oder aus Bäumen mit verschiedenen Hüllkörpern.[16]

### 2.3.3 Raumaufteilungen

Die Behandlung von großen Mengen an Objekten ist, wie bereits erwähnt, ein weiteres Problem der Kollisionserkennung. Mit der Raumpartitionierung kann die Anzahl der zu testenden Objektpaare für die Kollisionserkennung wesentlich reduziert werden. Die Idee dahinter ist, den Raum in einzelne Zellen aufzuteilen und nur noch Objekte in derselben Zelle auf Kollisionen zu untersuchen. Zu den klassischen Verfahren gehören die uniforme Raumunterteilung, Quadtrees, Oktrees, BSP-Trees (Binary Space Partitioning) und noch weitere Verfahren.[17]

**Uniforme Raumauteilung**

Bei dieser Methode wird ein gleichmäßiges Gitter im Kollisionsraum aufgespannt. Jedes Objekt wird mindestens einer Zelle zugeordnet. Die Kollisionserkennung muss nur noch zwischen Objekten durchgeführt werden, die sich in der gleichen Zelle befinden.

**Oktree/Quadtree**

Bei diesem Verfahren wird der gesamte Raum in eine Hierarchie aus achsenorientierten Quadern aufgeteilt. Die Wurzel besteht aus einem Quader, der den gesamten Raum enthält. In der nächsten Stufe wird der Quader weiter aufgeteilt. Dabei ist die Aufteilung von dem verwendeten Verfahren abhängig. Der Oktree z. B. teilt in jeder nächsten Stufe den Quader in acht kleinere Quader auf. Beim Quadtree entsprechend in vier kleinere Quader. Es müssen jetzt nur die Quader untersucht werden, in denen sich Objekte befinden. Der Vorteil zu der uniformen Raumaufteilung ist, dass die Zellengröße der Quader an die Größe der Objekte angepasst ist. Allerdings erhöht sich der Aufwand die Struktur zu aktualisieren, wenn sich Objekte bewegen.[18]

---

[16] Vgl. Eckstein (1998), S. 25 ff.
[17] Vgl. hierzu und im Folgenden Eckstein (1998), S. 129 ff.
[18] Vgl. Eckstein (1998), S. 130 ff.

### BSP-Trees

Bei diesem Verfahren wird der Raum durch Flächen von Polygonen aufgeteilt. Die Flächen werden in einem binären Baum eingeordnet. Jeder Knoten des Baums speichert die Ebene einer Fläche, die den Raum weiter aufteilt.[19]

### 2.3.4 Basiskollisionserkennung

Wie in den vorherigen Kapiteln schon beschrieben, kann durch die Verwendung von bestimmten Algorithmen die Anzahl der zu untersuchenden Objektpaare verkleinert werden. Kommt es aber tatsächlich zu Kollisionen zwischen Objekten müssen diese mit den Basiskollisionsalgorithmen getestet werden. Hierbei wird überprüft, ob sich zwei Flächenmengen schneiden. Dies ist genau dann der Fall, wenn es eine Kante oder Ecke der einen Fläche gibt, die die andere Fläche durchdringt. Ein Verfahren um dies zu testen ist z. B. die trennende Ebene. Zwei Körper berühren sich genau dann nicht, wenn eine trennende Ebene zwischen ihnen aufgespannt werden kann.[20]

## 2.4 Physik-Engines

Physik-Engines sind für die Simulation physikalischer Abläufe verantwortlich. Es sind einige kommerzielle wie auch nicht kommerzielle Engines verfügbar, die für verschiedenartige Einsatzgebiete konzipiert sind. Dazu gehören die Spielephysikengines und Physikengines für virtuelle Realitäten. Zu den kommerziellen Physik-Engines gehören **Havok Physics**, **Ageia PhysX** (Meqon), **RenderWare Physics und True Axis Physics** die alle für Windows entwickelt worden sind. Daneben gibt es zum einen die kostenlose Physik-Engine **Newton Game Dynamics**, lauffähig für Windows, Linux und Mac OS, sowie die beiden Open-Source-Engines **Tokamak** für Windows und die **Open Dynamics Engine** für Windows, Linux und Mac OS.

Wie schon in der Aufgabenstellung erwähnt, wird ODE als Basis für die eigene Simulationsbibliothek verwendet und im nächsten Kapitel genauer beschrieben.

---

[19] Vgl. Eckstein (1998), S. 131 ff.
[20] Vgl. Eckstein (1998), S. 93 ff.

## 2.5 Open Dynamics Engine

Die Open Dynamics Engine ist eine frei erhältliche, plattformunabhängige Bibliothek für die Simulation der Dynamik starrer Körper in einer virtuellen Umgebung von Russel Smith[21].

Die Physik-Engine gilt als schnell, leistungsstark, robust sowie flexibel und steht unter der BSD Lizenz. Sie kann daher nach eigenem belieben verändert und verbreitet werden.

ODE wird stetig weiterentwickelt und ist sehr gut dokumentiert. Bei der Simulation wird ein höchst stabiler Integrator eingesetzt, so dass Simulationsfehler nicht zu einem instabilen System führen können.

In ODE findet grundsätzlich eine Unterscheidung zwischen der Simulation der Dynamik starrer Körper und der Kollisionserkennung zwischen den Körpern statt. Dafür ist in ODE eine integrierte Kollisionserkennung enthalten, die auch Reibung unterstützt. Sie muss allerdings nicht zwingend eingesetzt werden. Zudem steht auch eine optimierte Kollisionserkennung namens OPCODE (**OP**timzed **CO**llision **DE**tection) zur Verfügung. Auf diese optimierte Variante wird hier hingegen nicht eingegangen.

Die Bibliothek bietet selbst keine Funktionalität für die Darstellung von visuellen Informationen an, daher wird für die Visualisierung der Simulation eine zusätzliche Bibliothek benötigt. In dem Software-Paket ist aber auch eine kleine Grafikbibliothek namens “Drawstuff“ enthalten, die auf OpenGL basiert und für eine Darstellung der Simulationen ausreicht. Die Abbildung 2-8 zeigt eine einfache Beispiel-Simulation die mit der Drawstuff-Bibliothek visualisiert wurde.[22]

Die Ausführungen in diesem Kapitel basieren alle auf dem ODE Benutzerhandbuch, auf das hiermit verwiesen wird. Eine detailliertere Beschreibung der Funktionsweise von ODE lässt sich dort nachlesen.[23]

---

[21] Siehe Smith (2006), http://www.q12.org.
[22] Siehe ODE Homepage. http://www.ode.org/, (2006).
[23] Siehe ODE User Guide. http://www.ode.org/, (2006).

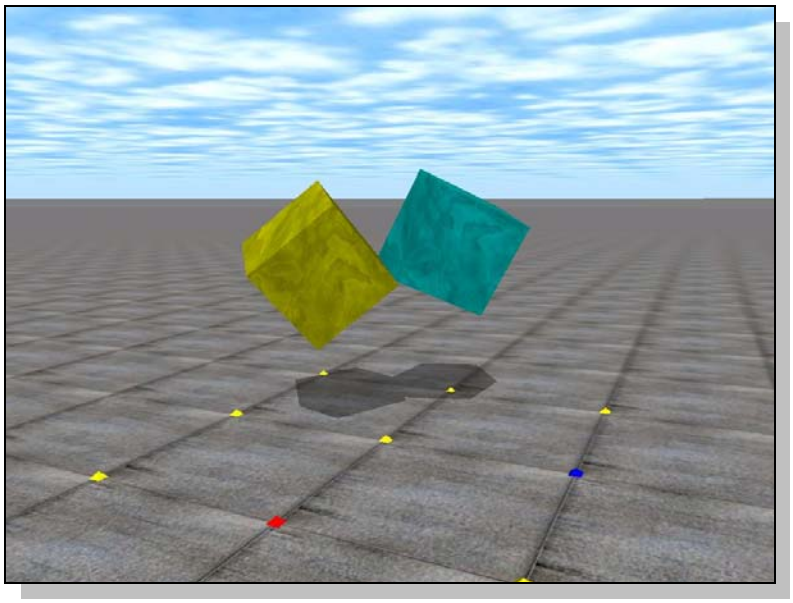


Abbildung 2-8: Simulation von zwei miteinander verbundenen Körpern, Quelle: HRI.

### 2.5.1 Struktur von ODE

ODE definiert für die Simulation starrer Körper und für die Kollisionserkennung fünf verschiedene Objekte, die in der Abbildung 2-9 schematisch dargestellt sind.

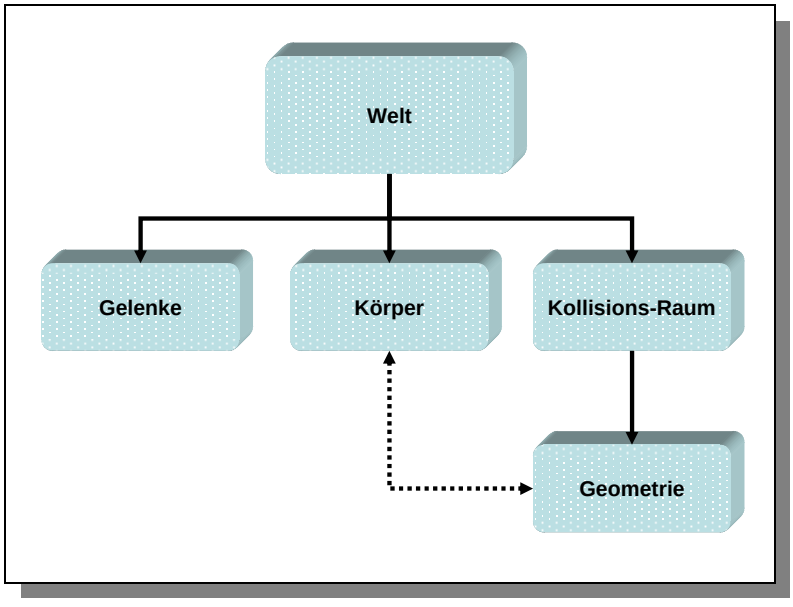



Abbildung 2-9: Schematische Darstellung der Objektstruktur in ODE, Quelle: Eigene Darstellung.

Das Objekt Welt ist die äußere Hülle, die alle anderen Objekte einschließt. In der Welt befinden sich die Körper, die über Gelenke miteinander verbunden werden können. Ein Körper definiert in ODE nur die dynamischen Eigenschaften (wie z. B. Geschwindigkeit, Masse usw.) eines physikalischen Objekts. Es werden keine Aussagen über die Form und das Aus-

sehen gemacht. Dafür wird jedem Körper eine Geometrie zugewiesen, was zum einen die äußere Form des Körpers definiert und zum anderen für die Kollisionserkennung nötig ist.

Es ist möglich die Welt in mehrere Kollisionsräume aufzugliedern (siehe Kapitel 2.3.3), um eine Optimierung der Kollisionserkennung zu erreichen. Ein Kollisionsraum besteht dabei aus mehreren Geometrie-Objekten. Körper sind nun indirekt durch die Zuweisung zu einem Geometrie-Objekt einem bestimmten Kollisionsraum zugeordnet. Allerdings ist es nicht zwingend erforderlich die Welt in solche Räume aufzuteilen, dies erspart jedoch einen erheblichen Rechenaufwand. Kollisionsräume sind auf jeden Fall in der Simulation einzusetzen, sobald mehrere Körper zu einem komplexen Kollisions-Objekt wie z. B. zu einem Automobil verbunden werden.

In der Abbildung 2-10 wird die Verbindung von mehreren Körpern mit Gelenken zu einem zusammenhängenden Objekt anschaulich dargestellt. Dadurch ist es möglich komplexere Strukturen aufzubauen. Hierfür stehen verschiedene Gelenktypen wie z. B. ein Scharnier zur Verfügung, die an späterer Stelle noch näher erläutert werden.

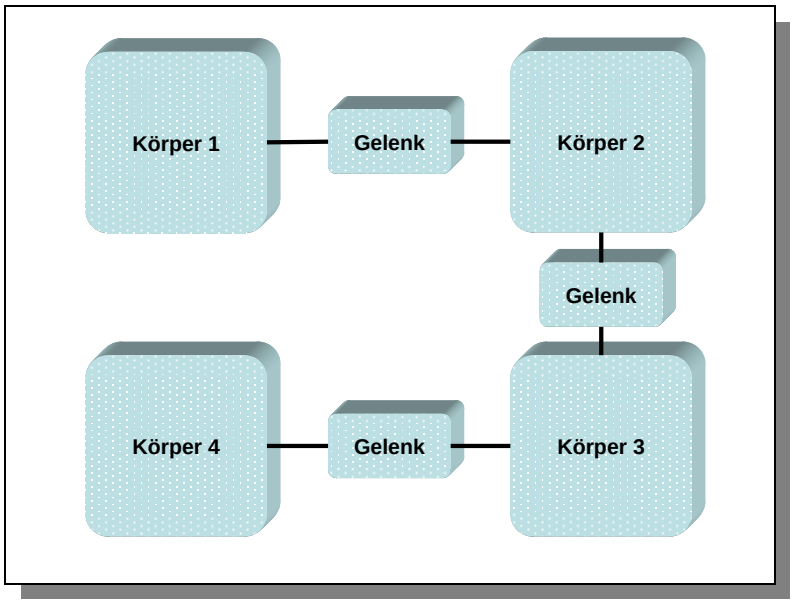



Abbildung 2-10: Verbindung von Körpern mit Gelenken, Quelle: Eigene Darstellung.

In der Abbildung 2-11 wird die Struktur einer virtuellen Welt auf Basis von ODE illustriert. Diesmal wird die Trennung von Welt und Kollisionsraum deutlich und die Beziehung zwischen Körpern und Geometrien veranschaulicht. Einem Körper können mehrere Geometrien zugeordnet werden. Dadurch lassen sich komplexere Kollisionsobjekte erstellen.

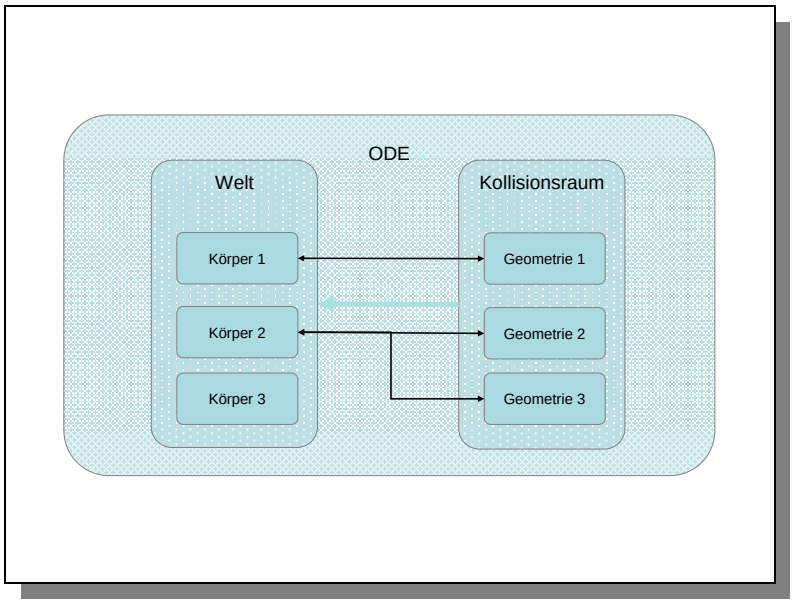



Abbildung 2-11: Struktur einer virtuellen Welt in ODE, Quelle: Eigene Darstellung.

### 2.5.2 Die Welt

Das Welt-Objekt symbolisiert die virtuelle Umgebung für die Simulation der Körperdynamik. Es stellt eine Art Behälter dar, in dem die Körper, die Verbindungen zwischen den Körpern (Gelenke), die Kollisionsräume und die Geometrie-Objekte gespeichert werden.

Es ist möglich mehrere virtuelle Welten zu erzeugen, die voneinander unabhängig sind. So können z. B. Körper-Objekte aus zwei verschiedenen Welten nicht miteinander kollidieren. In den meisten Anwendungen reicht es aus nur eine Welt zu erzeugen, sofern keine Simulationen zu unterschiedlichen Zeitpunkten stattfinden müssen. In diesem Fall ist eine weitere Welt pro Zeitpunkt erforderlich, da alle Objekte die sich in derselben Welt befinden immer nur zur gleichen Zeit existieren können.

Über das Welt-Objekt wird ein Simulationsschritt durchgeführt und es lassen sich verschiedene Einstellungen in der virtuellen Umgebung vornehmen, die sich auf jedes Objekte in der Welt beziehen. So kann z. B. eine Gravitationskraft hinzugefügt werden, die sich auf alle Objekte in gleicher Weise auswirkt.

Zur Durchführung eines Simulationsschritts in ODE stehen zwei verschiedene Möglichkeiten zur Verfügung. Die erste Methode **Step** führt einen Simulationsschritt aus und ist die genauere aber auch langsamere der beiden Methoden. Diese Methode verwendet ein System (eine große Matrix) von linearen (Un-)Gleichungen, die zu lösen sind. Dies hat einen Zeitfaktor der Ordnung $m^3$ und eine Speicherfaktor der Ordnung $m^2$, wobei $m$ die Anzahl der Bedingungen ist. Für große Simulationen kann das eine große Speicherbelastung bedeuten

gungen ist. Für große Simulationen kann das eine große Speicherbelastung bedeuten und die Berechnungen können beträchtlich lange dauern.

Die Funktion **QuickStep** führt einen Simulationsschritt mit Hilfe einer iterativen Methode durch. Diese Funktion hat einen Zeitfaktor der Ordnung $m*N$ und einen Speicherfaktor von $m$, wobei $m$ die Anzahl der Bedingungen ist und $N$ die Anzahl von Iterationen. Für große Simulationen ist dies die weitaus schnellere aber auch die ungenauere Methode.

### 2.5.3 Der Körper

Das Körper-Objekt symbolisiert einen starren Körper in der Simulation. Jeder Körper hat einen Referenzpunkt, der mit dem Massenschwerpunkt übereinstimmt (siehe Kapitel 2.2.1). Über einen Positionsvektor (x, y, z) wird die Position des Referenzpunktes bestimmt, dem auch eine lineare Geschwindigkeit (vx, vy, vz) zugewiesen werden kann. Die Orientierung eines Körpers ist durch eine 3*3 Matrix oder ein Quaternion gegeben. Ein Winkelgeschwindigkeitsvektor (wx, wy, wz) beschreibt dabei die Orientierung des Körpers über die Zeit hinweg. Weiterhin besitzt ein Körper die zeitkonstanten Eigenschaften Masse und eine Trägheitsmatrix, welche die Massenverteilung um den Massenschwerpunkt beschreibt.

Körper werden wie schon erwähnt über Gelenke miteinander verbunden. Eine Gruppe von Körpern, in der jeder Körper in irgendeiner Weise mit jedem anderen Körper aus einer Gruppe verbunden ist, wird als so genannte „Insel“ bezeichnet. Jede Insel in der Welt wird bei einem Simulationsschritt separat als eine Einheit behandelt.

Es besteht die Möglichkeit Körper zu deaktivieren und wieder zu aktivieren. Deaktivierte Körper werden bei einem Simulationsschritt nicht aktualisiert und sparen dadurch Berechnungszeit. Wenn also von einem Körper bekannt ist, dass er bewegungslos oder für die Simulation nicht relevant ist, sollte er deaktiviert werden. Bei einer großen Anzahl von Körpern ist es aber so gut wie unmöglich jeden Körper manuell zu überprüfen, ob er sich bewegt oder nicht. Daher bietet ODE eine Funktion an, bei der automatisch ein Körper deaktiviert wird, wenn er sich in mehreren Simulationsschritten oder in einer bestimmten Zeitspanne nicht bewegt hat. Ein Körper wird als unbeweglich betrachtet, sobald der Betrag der linearen Geschwindigkeit und der Winkelgeschwindigkeit unter einem bestimmten Grenzwert liegen. Wird ein deaktivierter Körper mit einem aktiven Körper über ein Gelenk verbunden, so wird er automatisch beim nächsten Simulationsschritt reaktiviert.

### 2.5.4 Das Gelenk

Ein Gelenk dient der Verbindung von Körper-Objekten. Es ist eine Beziehung bzw. Bedingung, die zwischen zwei Körpern aufgestellt wird. Dadurch können diese zwei Körper nur bestimmte Positionen und Orientierungen zueinander haben. Die folgenden Gelenktypen werden von ODE bereitgestellt: Kugel-Gelenk, Scharnier-Gelenk, Schiebe-Gelenk, Universal-Gelenk, Scharnier-Typ2-Gelenk, Kontakt-Gelenk. Es gibt noch zwei weitere Gelenktypen, das Fixier-Gelenk und das Motor-Gelenk, die aber nur in Spezialfällen zum Einsatz kommen. Im Folgenden sind die wichtigsten dieser Gelenke beschrieben.

**Kugel-Gelenk**

Die Abbildung 2-12 zeigt die Verbindung von zwei Körpern durch ein Kugelgelenk. Die beiden Körper können jegliche Position zueinander einnehmen, solange die Kugel des ersten Körpers sich in der Fassung des zweiten Körpers befindet.

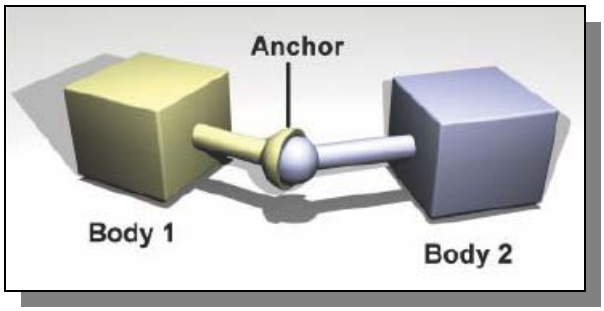



Abbildung 2-12: Kugel-Gelenk, Quelle: ODE User Guide.

**Scharnier-Gelenk**

Die Abbildung 2-13 zeigt die Verbindung von zwei Körpern über ein Scharniergelenk. Dieses Gelenk bedingt, dass sich beide Körper in derselben Position zueinander befinden und sich nur entlang der Scharnierachse bewegen können.

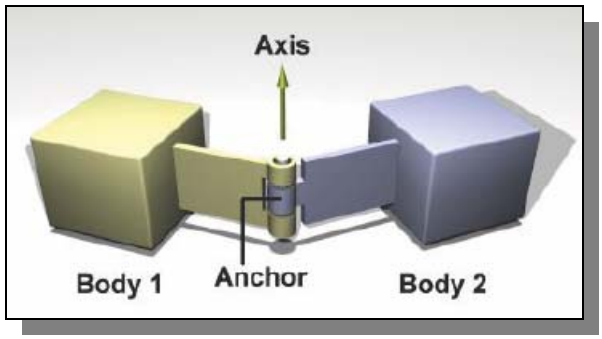



Abbildung 2-13: Scharnier-Gelenk, Quelle: ODE User Guide.

### Schiebe-Gelenk

Die Abbildung 2-14 zeigt die Verbindung von zwei Körpern durch ein Schiebegelenk. Dieses Gelenk bedingt, dass die beiden Objekte sich nur entlang einer Achse bewegen und die gleiche Orientierung zueinander haben.

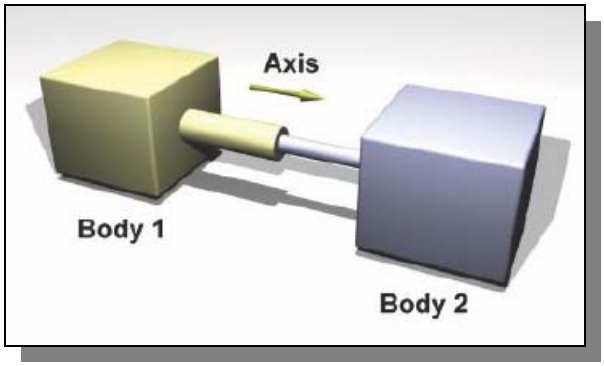



Abbildung 2-14: Schiebe-Gelenk, Quelle: ODE User Guide.

Die Abbildung 2-15 veranschaulicht die drei Gelenktypen: Universal-Gelenk, Scharnier-Typ2-Gelenk und ein Kontakt-Gelenk.

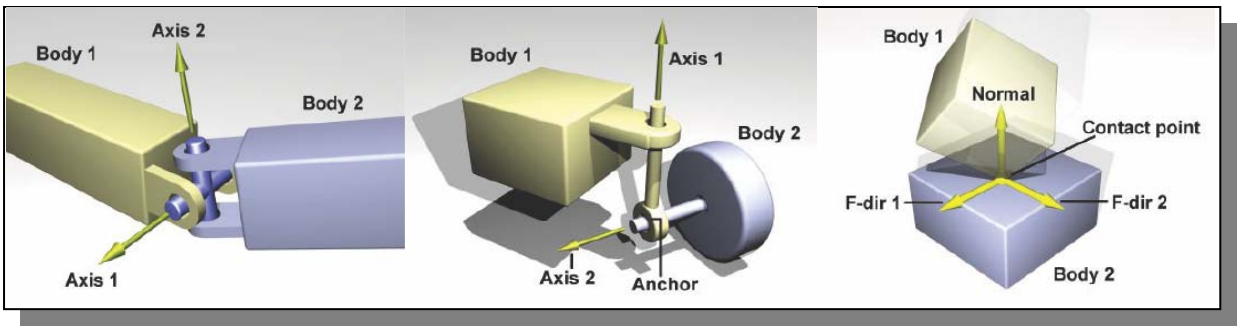



Abbildung 2-15: Universal-, Schanier-Typ2- und Kontakt-Gelenk, Quelle: ODE User Guide.

Bei jedem Simulationsschritt ist es den Gelenken möglich, Kräfte auf die mit ihnen verbundenen Körper zu übertragen. So können bei einer Bewegung des Körpers die Gelenkbedingungen eingehalten werden. Die Abbildung 2-16 macht dieses Prinzip deutlich. Dort sind zwei Körper mit einem Gelenk verbunden. Falls sich die Körper während eines Simulationsschritts auseinander bewegen, ist es dem Gelenk durch eine Kraft möglich dagegen zu wirken.

Jedes Gelenk besitzt mehrere Parameter, um z.B. die Position des Gelenks zu bestimmen. Die Parameter beziehen sich auf globale und nicht körperbezogene Koordinaten. Daher müssen die Körper bevor sie mit einem Gelenk verbunden werden zuerst in die richtige Position gebracht werden.

Für die leichtere Handhabung von Kontakt-Gelenken, die bei der Kollisionserkennung immer wieder neu entstehen, gibt es eine spezielle Gelenkgruppe. Dieser Gruppe werden alle Kontakt-Gelenke zugeordnet, damit sie nach jedem Simulationsschritt mit nur einem Funktionsaufruf wieder gelöscht werden können.

### 2.5.5 Gelenkfehler

Wenn ein Gelenk zwei Körper verbindet, müssen diese Körper eine bestimmte Position und Orientierung relativ zueinander haben. Dennoch ist es für die beiden Körper möglich sich in einer Position zu befinden, in der die Gelenkbedingungen nicht übereinstimmen. Dieser Fehler kann auf zwei Arten entstehen. Zum einen können während der Simulation Fehler auftreten, so dass die Körper von der gewünschten Position abweichen. Zum anderen ergeben sich Fehler, wenn der Anwender die Position und Orientierung eines Körpers nicht richtig setzt.

Es gibt einen Mechanismus diesen Gelenkfehler zu verringern. Während jedem Simulationsschritt wendet jedes Gelenk eine spezielle Kraft an, um die Körper zurück an ihre richtige Position zu bringen. Diese Kraft wird durch den ERP (**E**rror **R**eduction **P**arameter) kontrolliert. Der Parameter kann einen Wert zwischen 0 und 1 annehmen.

Der ERP Wert spezifiziert inwieweit der Gelenkfehler behoben wird. Bei einem Wert von 0 wird keine Berichtigungskraft angewandt und die Körper könnten eventuell auseinander driften. Bei einem Wert von 1 wird versucht, alle Gelenkfehler auszubessern. Dennoch ist es nicht ratsam einen ERP Wert von 1 zu benutzen, da der Fehler nicht vollständig behoben wird. Daher wird ein Wert zwischen 0,1 und 0,8 empfohlen. Der ERP Wert kann global für eine Welt gesetzt werden. Es ist aber auch möglich jedem Gelenk einen spezifischen Wert zuzuweisen.

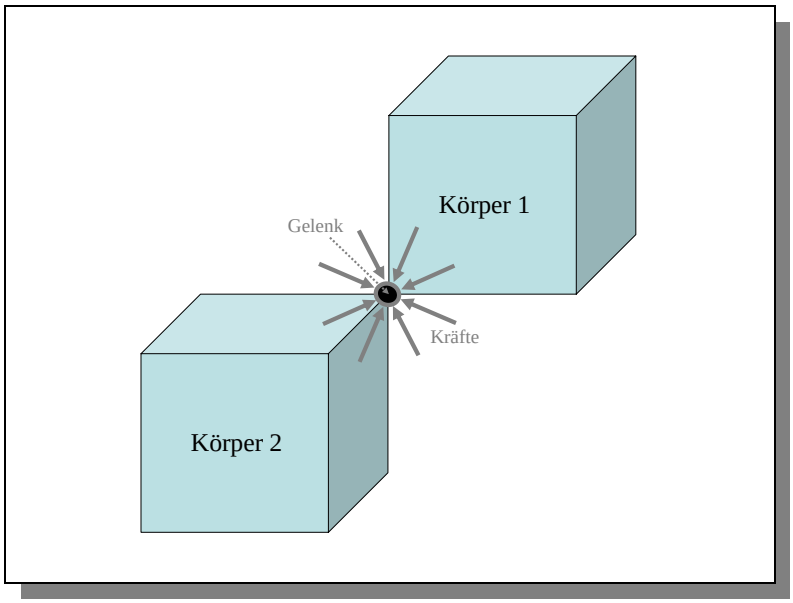



Abbildung 2-16: Funktionsweise des Error Reduction Parameters, Quelle: Eigene Darstellung.

### 2.5.6 Weiche Bedingungen

Die Bedingungen die durch ein Gelenk zwischen zwei Körpern aufgestellt werden, sind fest und dürfen nicht gebrochen werden. Es ist aber in manchen Fällen nötig, dass die Gelenke nicht vollkommen starr sind, sondern einen elastischen Charakter aufweisen. Dies ist z. B. bei Kontakt-Gelenken von Vorteil, bei denen die Kollisionsfläche nachgeben soll, um etwa weiches Material zu simulieren. Dafür gibt es neben dem ERP den CFM-Parameter (**C**onstraint **F**orce **M**ixing), durch den die Starrheit des Gelenks bestimmt wird. Ein Wert von 0 bedeutet, dass die Bedingung hart ist. Ein positiver Wert ermöglicht, dass die Bedingung übertreten werden darf.

### 2.5.7 Einsatz von CFM und ERP

Die Kombinationen des CFM- und ERP-Parameters macht es möglich, verschiedene Effekte zu erreichen. So kann z. B. jede gewünschte Feder-Dämpfungs-Bedingung in einem Gelenk nach den folgenden Formeln berechnet werden:

$$ERP = \frac{h * k_p}{(h * k_p + k_d)}$$

$$CFM = \frac{1}{(h * k_p + k_d)}$$

$$k_p = Federkonstante\,,\ k_d = Dämpfungskonstante\,,\ h = Schrittgröße$$

### 2.5.8 Reibung

ODE unterstützt die Modellierung von Reibung an Kontaktpunkten. Dafür verwendet es das einfache Reibungsmodell von Coulomb. Die Regel lautet:

$$\left|F_T\right| \leq \mu * \left|F_N\right|$$

Wobei $F_T$ die Tangentialkraft und $F_N$ die Normalkraft ist. $\mu$ ist der Reibungskoeffizient.

### 2.5.9 Kollisionsräume

Ein Kollisionsraum ist ein spezielles Geometrie-Objekt. Es kann als ein Behälter angesehen werden, indem sich weitere Geometrie-Objekte befinden. Somit ähnelt es sehr dem Welt-Objekt bei der Simulation der Dynamik, nur mit dem Unterschied das hier keine Dynamik sondern Kollisionen simuliert werden. Die Kollisionsräume wurden eingeführt, um die Kollisionserkennung zu beschleunigen (siehe Kapitel 2.3). Ohne diese Räume müssten alle Objektpaare, die in der Welt bestehen überprüft werden, ob sie sich gegenseitig berühren. Damit ist bei einer hohen Anzahl von Objekten ein sehr großer Rechenaufwand verbunden. Eine bessere Vorgehensweise ist, jede Geometrie einem Kollisionsraum hinzuzufügen und dann eine Kollisionserkennung davon abhängig zu machen, in welchem Kollisionsraum es sich befindet.

### 2.5.10 Die Geometrie-Objekte

Die Kollisionserkennung basiert vor allem auf den Geometrie-Objekten. Sie bestimmen die Form und Gestalt eines Körpers und haben im Gegensatz zu starren Körpern keine dynamischen Eigenschaften, wie z. B. Masse oder Geschwindigkeit, sondern geometrische Werte wie Größe, Position, Form usw.. Es gibt zwei verschiedene Arten von Geometrie-Objekten. Diejenigen, die Ihre Orientierung und Position während der Simulation verändern können und solche die das nicht können, wie z. B. die statische Umgebung.

Um die Kollisionserkennung in der Dynamik-Simulation zu verwenden, muss das Geometrie-Objekt mit einem starren Köper-Objekte assoziiert werden. ODE unterstützt sechs verschiedene Typen von Geometrie-Objekten, wie in der Abbildung 2-17 dargestellt. Dazu gehören Kugel, Quader, Zylinder, Strahl, Dreieck-Maschen und benutzerdefinierte Geometrien.

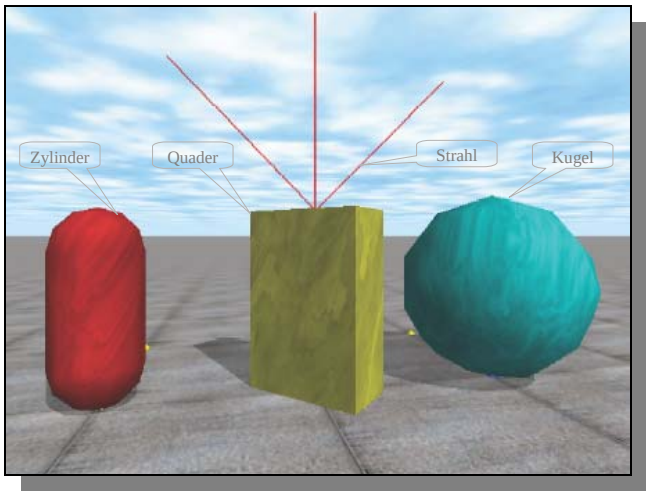



Abbildung 2-17: Die verschiedenen Geometrien in ODE, Quelle: HRI.

Wie die Abbildung 2-18 veranschaulicht, können einem Körper mehrere Geometrien zugewiesen werden. Dies erlaubt die Konstruktion von wesentlich komplexeren Objekten. Die Trennung von Körper- und Geometrie-Objekte hat dabei einige Vorteile. So ist die Erstellung von unsichtbaren Objekten möglich, die einem anderen Objekt wiederum als zusätzliches Gewicht hinzugefügt werden können. Es wird lediglich ein Körper erzeugt, ohne diesem eine Geometrie zuzuweisen. Umgekehrt ermöglicht die Erstellung von Geometrien, ohne sie einem Körper zuzuweisen, die Erzeugung von unbeweglichen Objekten.

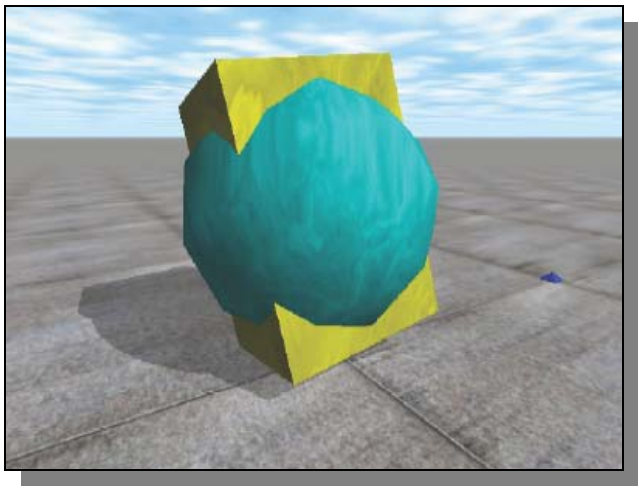


Abbildung 2-18: Körper in ODE mit mehreren Kollisionsobjekten, Quelle: HRI.

### 2.5.11 Kollisionserkennung

ODE besteht wie bereits erwähnt aus zwei Hauptkomponenten: Zum einen aus der Bibliothek für die Simulation der Dynamik starrer Körper und zum anderen aus der Bibliothek für die Kollisionserkennung. Die in ODE integrierte Kollisionserkennung muss nicht zwingend verwendet werden. Es ist auch möglich andere Bibliotheken zu benutzen, so lange sie die

richtigen Informationen für die zu generierenden Kontaktpunkte liefern. Die Bibliothek unterstützt zurzeit die oben angegebenen Kollisions-Objekttypen.

Die Bibliothek für die Kollisionserkennung gibt im Gegensatz zur Dynamik-Bibliothek Informationen über die Form, Gestalt und Geometrie der Körper. Vor jedem Simulationsschritt wird untersucht, welche Körper sich berühren, um anschließend die entsprechenden Kontaktpunkte zu übergeben. An diesen Kontaktpunkten werden dann Verbindungen (Kontakt-Gelenke) erstellt. Ein Kontaktpunkt hat dabei eine ganz spezielle Struktur. Diese Struktur besteht aus einem Positionsvektor, einem Normalenvektor, der Durchdringungstiefe und den zwei beteiligten Geometrie-Objekten. Dies kann auch die statische Umgebung sein.

Bevor die Kollisionserkennung genutzt werden kann ist zunächst eine Kollisionswelt ähnlich der Simulationswelt zu erzeugen, in der die Kollisionserkennung stattfindet. Anschließend sind dann alle für die Simulation benötigten Geometrie-Objekte zu erstellen und der Kollisionswelt hinzuzufügen. Vor dem Start der Kollisionserkennung, ist noch ein Behälter für die Kontaktverbindungen bereitzustellen. Dieser wird in ODE als “joint-group“ bezeichnet und speichert alle während der Kollisionserkennung erstellten Kontaktverbindungen in einer Gruppe, damit sie nach dem Simulationsschritt auf einer einfacher Weise wieder gelöscht werden können.

**Ablauf der Kollisionserkennung**

Unten stehende Ausführungen beschreiben die genaue Vorgehensweise bei der Kollisionserkennung:

1. Vor jedem Simulationsschritt wird durch eine spezielle Funktion überprüft, welche Körper sich berühren und eine Liste mit den entsprechenden Kontaktpunkten zurückgeliefert.
2. Für jeden Kontaktpunkt wird eine spezielle Kontaktverbindung erstellt, die Informationen zur Reibung und weitere Eigenschaften enthält.
3. Alle erstellten Kontakte werden in einer speziellen Gruppe gespeichert, um sie nach dem Simulationsschritt auf schnellen Weg wieder zu löschen.
4. Ausführen des Simulationsschritts.
5. Alle Kontakte werden gelöscht.

ODE stellt für die Kollisionserkennung drei verschiedene Funktionen bereit, die eine unterschiedliche Behandlung der Kollisionserkennung ermöglichen. Im Folgenden sollen diese beschrieben werden:

### 1. Collide

Die Funktion Collide bekommt zwei sich möglicherweise überschneidende Geometrie-Objekte übergeben und generiert bei einer Überschneidung Kontaktpunkte zwischen den Objekten. Wenn sich die Objekte nicht berühren wird der Wert Null zurückgeliefert, andernfalls die Anzahl der Kontakte. Diese Funktion macht keinen Unterschied, ob es sich bei den übergebenden Objekten um Kollisionsräume handelt oder einzelne Geometrien. Wenn eines oder beide der übergebenden Objekte ein Kollisionsraum ist, werden alle Objekte des einen Kollisionsraumes mit allen Objekten des anderen Kollisionsraums verglichen, sowie alle Kontaktpunkte zurückgegeben. Da bei dieser Funktion kein Unterschied zwischen Kollisionsräumen und Geometrie-Objekten gemacht wird, ist es besser die Funktionen SpaceCollide und SpaceCollide2 zu verwenden, die solch eine Kontrolle ermöglichen.

### 2. SpaceCollide

Die Funktion SpaceCollide erzeugt die Kontaktpunkte nicht direkt, sondern bestimmt zunächst einmal alle Geometrie-Objekte innerhalb eines Kollisionsraums, die sich möglicherweise überschneiden. In einer Callback-Funktion, welches die beiden sich möglicherweise überschneidenden Objekte übergeben bekommt, kann nun für jedes Objektpaar genau bestimmt werden, ob die Funktion Collide zur Kontakt-Generierung aufgerufen wird. In dieser Callback-Funktion ist es zusätzlich möglich, die genauen Kontakt-Parameter für jeden Kontakt einzustellen.

### 3. SpaceCollide2

Die Funktion SpaceCollide2 ist der Funktion SpaceCollide ähnlich, nur mit der Ausnahme dass sie Geometrie-Objekte aus einem Kollisionsraum mit Geometrie-Objekten aus einem anderen Kollisionsraum nach möglichen Überschneidungen überprüft. Das genaue Verhalten der Funktion ist abhängig vom Typ der übergebenen Geometrien.

1. Wenn eines der Argumente ein Kollisionsraum spezifiziert und das andere nur ein Geometrie-Objekt, wird die Rückruffunktion für alle möglichen Überschneidungen zwischen dem Geometrie-Objekt und den Objekten in dem übergebenem Kollisionsraum aufgerufen.
2. Wenn beide übergebene Argumente Kollisionsräume sind, wird die Rückruffunktion für alle sich möglich überschneidenden Objektpaare aufgerufen, wobei eines der Objekte dem ersten Kollisionsraum angehören muss und das andere Objekt dem zweiten Kollisionsraum.

ODE stellt zurzeit drei verschiedene Arten von Kollisionsräumen zur Verfügung. Den Simple Space, den Multiresolution Hash Table Space und den QuadTree Space (siehe hierzu Kapitel 2.3.3). Alle drei Kollisionsräume verwenden unterschiedliche Strukturen für die Speicherung der Geometrie-Objekte und benutzen unterschiedliche Algorithmen zur Kollisionserkennung.

**1. Simple Space**

Hier wird keine optimierte Kollisionserkennung angewendet. Es werden alle möglichen Objektpaare auf Überschneidungen überprüft (siehe Kapitel 2.3.4). Die Zeit, die für die Überschneidungstests für $n$ Objekte benötigt wird, ist $O(n^2)$ [24]. Daher ist es für Simulationen mit einer großen Anzahl an Objekten nicht so gut geeignet. Allerdings ist es für kleinere Simulationen mit wenigen Objekten die bevorzugte Variante.

**2. Multiresolution Hash Table Space**

Hier wird eine interne Datenstruktur benutzt, die den Kollisionsraum in ein dreidimensionales Raster aufteilt (siehe Kapitel 2.3.3). Jede Zellen-Überlappung wird protokolliert, wobei eine Zelle die Form eines Würfels mit der Seitenlänge $2^i$ hat. Die Zeit, die für Überschneidungstests für $n$ Objekte benötigt wird, beträgt $O(n)$, da jedes Objekt schnell mit den umliegenden Objekten überprüft werden kann. Voraussetzung ist, dass die Objekte nicht zu dicht aneinander angeordnet sind.

**3. QuadTree Space**

Hier wird ein hierarchischer rasterbasierter AABB-Baum (**A**xis-**A**ligned **B**ounding **B**ox) verwendet, um die Kollisionserkennung zu optimieren (siehe Kapitel 2.3.2). Es ist außergewöhnlich schnell für Simulationen mit großer Anzahl an Objekten im offenen Gelände.

### 2.5.12 Simulation / Integration

Der Prozess ein System von starren Körpern über die Zeit hinweg zu simulieren, wird Integration genannt. Jeder Integrationsschritt erhöht die aktuelle Zeit durch eine vorgegebene Schrittgröße und passt den Status von allen Körpern der neuen Zeit an. Es wird ein Integrator erster Ordnung eingesetzt.

---

[24] Das Landau-Symbol O(N) wird in der Informatik dazu verwendet, um die Komplexität von Algorithmen zu spezifizieren. Vgl. Myers Enzyklopädisches Lexikon (1977), Band 14 S. 584.

Der Integrator, den ODE verwendet, ist sehr stabil, aber nicht besonders genau, außer bei einer kleinen Schrittgröße. Es ist geplant in einer späteren Version von ODE einen Integrator höherer Ordnung einzusetzen, mit dem genauere Simulationen durchgeführt werden können.

Zwischen jedem Integrationsschritt kann der Benutzer Funktionen aufrufen, um Kräfte an den Körpern anzuwenden. Diese Kräfte werden einem Kräftespeicher in einem Körper zugefügt. Wenn der nächste Integrationsschritt erfolgt, wird die Summe aller Kräfte auf den Körper angewendet. Anschließend wird der Kräftespeicher wieder gelöscht.

Die Abbildung 2-19 zeigt den generellen Aufbau einer Simulations- und Kollisionswelt sowie den genauen Ablauf der Dynamik-Simulation.

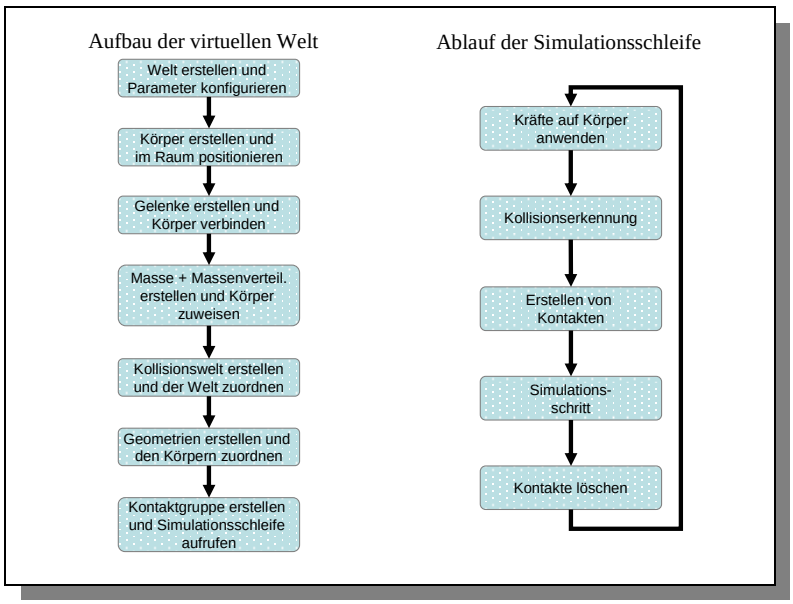

Aufbau der virtuellen Welt
Welt erstellen und Parameter konfigurieren
Körper erstellen und im Raum positionieren
Gelenke erstellen und Körper verbinden
Masse + Massenverteil. erstellen und Körper zuweisen
Kollisionswelt erstellen und der Welt zuordnen
Geometrien erstellen und den Körpern zuordnen
Kontaktgruppe erstellen und Simulationsschleife aufrufen
Ablauf der Simulationsschleife
Kräfte auf Körper anwenden
Kollisionserkennung
Erstellen von Kontakten
Simulations-schritt
Kontakte löschen

Abbildung 2-19: Aufbau und Ablauf einer Simulation in ODE, Quelle: Eigene Darstellung.

Auf der linken Seite der Abbildung 2-19 wird der Aufbau der virtuellen Welt dargestellt. Am Anfang werden zunächst die Welt und danach die einzelnen Körper erstellt. In der Welt lassen sich nun Parameter, wie z. B. die ERP- und CFM-Werte, einstellen und eine allgemeine Gravitationskraft spezifizieren. Für die Körper muss eine Masse und eine Massenverteilung, sowie die Position im Raum angegeben werden. Anschließend können die Gelenke zur Verbindung der Körper angelegt und entsprechend eingestellt werden. Zum Schluss ist die Kollisionswelt mit den entsprechenden Geometrien für die einzelnen Körper zu erzeugen. Diese sind noch den Körpern zuzuweisen. Bevor nun die Simulationsschleife aufgerufen wird, muss noch eine Gelenkgruppe für die Kontakt-Gelenke spezifiziert werden.

Auf der rechten Seite der Abbildung wird der Ablauf der Simulationsschleife dargestellt. Hier können z. B. Kräfte auf die Körper addiert werden. Im nächsten Schritt findet die Kollisions-

erkennung statt. Auf Basis der Kollisionserkennung werden zwischen den Körpern, die sich berühren oder zwischen Körpern und der statischen Umgebung, Kontakte erstellt, so dass sie sich nicht durchdringen können. Anschließend wird der eigentliche Simulationsschritt ausgeführt und die Dynamik der Körper berechnet. Am Ende müssen alle Kontaktverbindungen wieder gelöscht werden.

### 2.5.13 Simulationsprogramm

Dieser Abschnitt soll kurz den generellen Aufbau eines Simulationsprogramms in ODE dokumentieren. Zunächst wird in der Main-Funktion die virtuelle Welt mit allen darin enthaltenen Objekten erzeugt. Dazu gehört die Erstellung einer Kollisionswelt mit den entsprechenden Kollisionsobjekten und die Verbindung der Körper mit Gelenken. An dieser Stelle können auch verschieden Einstellungen vorgenommen werden, wie z. B. das Einstellen einer allgemeinen Gravitationskraft, die in der Welt auf alle Objekte wirken soll.

Nach der Konstruktion der virtuellen Welt wird der Grafikroutine die Simulationsroutine als Funktionszeiger übergeben. Die Grafik-Bibliothek ist nun dafür verantwortlich, dass in bestimmten Intervallen die Simulationsberechnung immer wieder aufgerufen wird. Innerhalb dieser Simulationsroutine erfolgt auch die Abfrage, ob es in der Simulation zu Kollisionen gekommen ist. Im Fall einer möglichen Kollision wird die Callback-Funktion aufgerufen, in der die eigentliche Kollisionsbehandlung stattfindet. Nach der Behandlung von Kollisionen wird nun der eigentliche Simulationsschritt ausgeführt, in der die Berechnung der Dynamik erfolgt. Die Simulation wird solange durchgeführt, bis der Benutzer sie durch ein bestimmtes Abbruchkriterium beendet.

## 2.6 ViToSA

Die Simulation der Dynamik von starren Körpern soll durch eine unabhängige Anwendung visualisiert werden. Dies erfolgt mit dem Programm ViToSA (**Vi**sualisation **To**ol for **S**imulation **A**pplications), ein Werkzeug für die Visualisierung von 3D-Informationen.
Das Programm ist eine von HRI entwickelte Anwendung, die in Qt[25] erstellt wurde und als allgemeines 3D-Visualisierungswerkzeug dient. Ein wesentliches Merkmal von ViToSA ist die Erweiterbarkeit des Programms.

[25] Qt ist eine Softwarebibliothek von Trolltech für die Erstellung von grafischen Benutzeroberflächen unter C++.

Die Abbildung 2-20 zeigt einen Screenshot des Programms. Im unteren Bereich der Abbildung befinden sich mehrere Registerkarten die jede für sich verschiedene Einstellungsmöglichkeiten bietet. Über die erste Registerkarte **Converters** lassen sich alle zur Verfügung stehenden Konverter in das Programm laden, entfernen und konfigurieren. Die nächste Registerkarte **Viewer** stellt Funktionen für das Betrachtungsfenster bereit, wie z. B. die Anzeige von Schatten, Achsen, Raster usw. Der Abschnitt **Camera** bietet Funktionen für die Kameraeinstellung, wie z. B. die Ansicht der Szene aus vordefinierten Positionen wie *Top*, *Front*, *3D* usw.. Der Szene können hier auch weitere Kameras an selbst definierten Positionen hinzugefügt werden. **Filter** ist die Registerkarte, die in der Abbildung 2-20 angezeigt wird. In dieser Ansicht werden alle Objekte der Szene in einer Baum-Struktur mit Angabe der Position aufgeführt. Der Bereich **Measurements** bietet eine Funktion, mit der Abstände zwischen ausgewählten Objekten aus der Szene gemessen werden können. Über das letzte Fenster können einzelne Bilder oder Bildsequenzen von der Szene generiert werden.

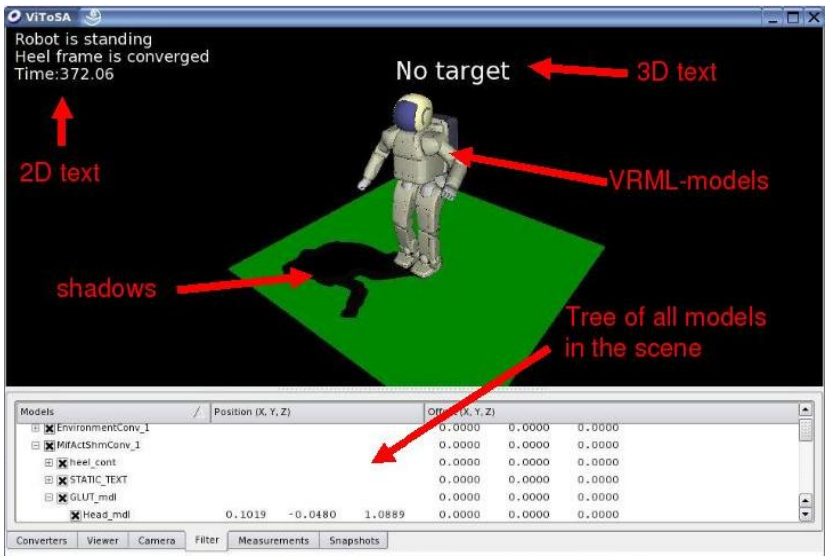



Abbildung 2-20: Screenshot von ViToSA, Quelle: HRI Internal Report, M. Mühlig (2006).

Die Anbindung von externen Programmen an ViToSA wird durch das Konzept eines Konverters ermöglicht. Ein Konverter ist ein in QT erstelltes Plug-In, der die anwendungsspezifischen Daten in entsprechende Nachrichten für ViToSA umwandelt. ViToSA verwendet intern eine eigene Datenstruktur **ObjectData**, mit der es die Objekte visualisiert. Über den in ViToSA eingesetzten Sourcehandler ist es möglich mehrere Konverter gleichzeitig zu laden, wobei die Daten der entsprechenden Anwendung alle in derselben Szene visualisiert werden. Der Konverter ist für jede Anwendung neu zu implementieren. Die Abbildung 2-21 veranschaulicht das Prinzip in einer Grafik.

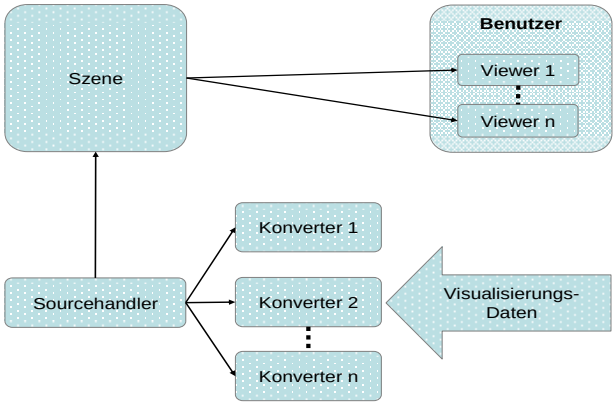



Abbildung 2-21: Konzeption von ViToSA, Quelle: HRI Internal Report, M. Mühlig (2006).

Der Ablauf der Visualisierung sieht nun folgendermaßen aus: Zunächst müssen dem Konverter die anwendungsspezifischen Daten übermittelt werden. Wie diese Übertragung der Daten an den Konverter erfolgt, ist in ViToSA nicht festgelegt und Aufgabe des Benutzers. Der Konverter muss nun die erhaltenen Informationen in ViToSA konforme Nachrichten umwandeln und sie an den Sourcehandler übermitteln. Der Sourcehandler stellt sicher, dass die umgewandelten Daten an die Szene gesendet und dort dargestellt werden. Dies ist auch für mehrere Konverter parallel möglich, wobei die Anzahl an Konvertern nicht beschränkt ist. Der Benutzer kann nun über den so genannten Viewer die Szene Betrachten. ViToSA bietet hier auch die Funktion mehrere Viewer zu benutzen.

# 3 Entwicklung eines Software-Simulationssystems

Dieses Kapitel umfasst den gesamten Entwicklungsprozess von der Planung des Simulationssystems bis zur Implementierung und Testen der Bibliotheken und der Test-Anwendungen. Es gliedert sich in die folgenden Unterkapitel:

1. Projektplanung
2. Anforderungen an das System
3. Simulationssystem
4. Simulator
5. Visualisierung
6. Manipulation
7. Testszenario

## 3.1 Projektplanung und Vorbereitung

Das Projekt ist in drei Phasen eingeteilt. In der ersten Phase findet die Einarbeitung und Vorbereitung statt. Die zweite Phase beschäftigt sich mit der Umsetzung des Projekts, die im Wesentlichen durch die Kapitel Simulationssystem, Simulator, Visualisierung und Manipulation beschrieben werden. In der letzten Phase findet die Dokumentation mit einem abschließenden Vortrag über das Ergebnis statt.

Am Anfang des Projekts wird der genaue zeitliche Ablauf des gesamten Vorhabens nach einer kurzen Einarbeitung in das Thema in einem Projektplan (siehe Abbildung 3-1) festgehalten. Nachdem sich ein gewisser Überblick verschafft wurde, soll in dieser Zeit auch ein Vortrag über das zu entwickelnde Simulationssystem stattfinden. Dies hat den Zweck Unklarheiten schon zu Beginn des Projekts ausfindig zu machen und Anregungen für die Realisierung zu geben.

Die Programmierung des gesamten Quellcodes erfolgt in der Sprache C und anhand der bei HRI geltenden Programmier-Konventionen. Die Übersetzung der entwickelten Programme erfolgt mit gcc (GNU C Compiler) und dem HRI internen Makefile-System. Zur Fehlersuche wird der Linux-Debugger DDD (Data Display Debugger) eingesetzt. Die Dokumentation der Software bzw. des Quellcodes erfolgt mit Doxygen, einem Open-Source Software-Dokumentationswerkzeug. Es steht als freie Software unter der GPL zur Verfügung. Die Versionskontrolle übernimmt die ebenfalls Open-Source Software SVN (Subversion).

| *Tätigkeit* | *Dauer (Tage)* | *Von* | *Bis* |
|---|---|---|---|
| **Projektvorbereitungen** | **25** | **01.05** | **02.06** |
| Einweisung in das Thema | | | |
| Projektplanung | | | |
| Einarbeitung in ODE | | | |
| Vortrag | | | |
| **Projektumsetzung** | **64** | **05.06** | **31.08** |
| Konzept für Simulationssystem | | | |
| Implementierung des Simulators | | | |
| Implementierung der Visualisierung | | | |
| Implementierung der Benutzerschnittstelle | | | |
| Testszenarios | | | |
| **Projektabschluss** | **21** | **01.09** | **29.09** |
| Dokumentation (Puffer) | | | |
| Abschlussvortrag | | | |
| **Dauer Insgesamt (Tage)** | **110** | **01.05** | **29.09** |

Abbildung 3-1: Projektplan.

## 3.2 Anforderungen

### 3.2.1 Anforderungen an das Simulationssystem

Es ist eine Software zu entwickeln, die in der Lage ist, eine virtuelle, dynamische Welt zu erstellen und zu simulieren. Diese Welt soll die Umgebung eines Roboters nachbilden, ohne den Roboter aber selbst zu simulieren, da ein Roboter-Simulator bereits existiert.

Damit die virtuelle Umgebung so real wie möglich wirkt, müssen die physikalischen Gesetze der realen Welt, also die Modelle von Massenträgheit, Reibung und Körperkollisionen, berücksichtigt (simuliert) werden.

Die Dynamik einzelner Objekte in der virtuellen Welt muss abschaltbar sein, um sie durch eine kinematische Steuerung von außen zu ersetzen. Dadurch soll der Eingriff durch den simulierten Roboter in die virtuelle Umgebung ermöglicht werden.

Eine weitere Anforderung an das System, ist die Möglichkeit in die Simulation interaktiv einzugreifen, d.h. neben der direkten Manipulation der virtuellen Welt durch den Simulator selbst (also durch die API) muss auch die Möglichkeit bestehen, von außen die aktuelle Simulation zu beeinflussen. Die Manipulation der virtuellen Welt durch eine externe Anwen-

dung kann dabei entweder über eine (Space-) Mouse oder durch eine einfache Tastatureingabe erfolgen.

Das System muss zudem eine Visualisierung bereitstellen, mit der die symbolischen Daten des Simulators dargestellt werden können.

Für die Kommunikation zwischen Simulator, Visualisierung und Manipulation sind die nötigen Schnittstellen bereitzustellen.

### 3.2.2 Anforderung an den Simulator

Der zu entwickelnde Simulator muss ganz bestimmte Anforderungen erfüllen, damit er für den Einsatz im späteren Simulations-System geeignet ist. Diese sind hier in funktionale und qualitative Anforderungen gegliedert.

**Funktionale Anforderungen**

- Zunächst muss generell die Möglichkeit gegeben sein, eine virtuelle Welt zu konstruieren, welche zunächst nur aus symbolischen Informationen besteht. Dies beinhaltet die Bereitstellung von primitiven Objekten, wie z.B. Quader, Kugel oder Zylinder, aus denen sich wiederum durch entsprechende Funktionalität (Gelenke) komplexere Strukturen erzeugen lassen.
- An den Objekten müssen Kräfte angesetzt werden können, die bei den Objekten zu einer Bewegung bzw. Beschleunigung führen.
- Eine Kollisionserkennung und -antwort ist notwendig, damit sich Objekte nicht durchdringen können, sondern aneinander abprallen bzw. anstoßen.
- Es ist eine dynamische und kinematische Simulation der Objekte zu ermöglichen.
- Es muss die Möglichkeit bestehen, einzelne Objekte von der Simulation auszuschließen (zu deaktivieren), d.h. dass an diesen Objekten keine Kräfte wirken. Sie müssen aber trotzdem mit den anderen Objekten interagieren können. Wenn z.B. ein an der Simulation teilhabendes Objekt auf ein nicht teilnehmendes Objekt trifft, darf es dieses nicht durchdringen, sondern muss auf dieses Objekt reagieren können.
- Die Manipulation von bestimmten Objekten, wie z. B. das Ändern von Position, Kräften, Geschwindigkeit usw., ist von außen durch geeignete Schnittstellen bereitzustellen.
- Der Status einzelner Objekte in der Simulation, wie z. B. Position, angreifende Kräfte, Geschwindigkeit usw., muss abfragbar sein.

- Der Simulator muss dafür die Dynamik von starren Körpern berechnen können, d. h., die am Körper anliegenden Kräfte müssen in eine Änderung des Weges, der Geschwindigkeit und der Beschleunigung umgesetzt werden.
- Die Visualisierung spielt für den Simulator direkt keine Rolle und sollte möglichst unabhängig von der Simulation ablaufen. Es müssen aber die entsprechenden Daten für eine mögliche Veranschaulichung der Simulation bereitgestellt werden.
- Die Initialisierung virtueller Welten soll mit Hilfe einer Konfigurationsdatei (Textdatei) erfolgen, damit eine dynamische Konfiguration des Simulators möglich ist.

**Qualitative Anforderungen**

Die qualitativen Anforderungen an das System sind zum einen eine schnelle physikalische Simulationsumgebung, die sich durch Robustheit und Absturzsicherheit, also einen stabilen Integrator, auszeichnet und zum anderen sollte die Simulation der Dynamik möglichst genau sein. Allerdings ist dies nicht immer zu gewährleisten, da durch den interaktiven Eingriff in die virtuelle Welt es immer wieder zu kleinen Ungenauigkeiten kommen kann.

### 3.2.3 Anforderung an die Manipulation

Für die Steuerung bzw. Beeinflussung des Simulators ist eine Schnittstelle zu implementieren, mit der es möglich ist, interaktiv in die Simulation einzugreifen und somit den Roboter an die Umgebungssimulation zu koppeln. Die Manipulation sollte aber auch durch eine externe Anwendung, z. B. eine Benutzerschnittstelle möglich sein. Dadurch wird der Anwender in die Lage gebracht, manuell in die Simulation einzugreifen. Dies kann z. B. durch eine (Space-) Mouse oder textbasiert durch die Tastatur erfolgen.

### 3.2.4 Anforderung an die Visualisierung

Für die Betrachtung der virtuellen Welt und der Abläufe, die in der virtuellen Umgebung stattfinden, wird eine Visualisierung benötigt, die eine dreidimensionale Darstellung von Roboter und Umgebung ermöglicht sowie virtuelle Kommandos anzeigen kann. Die Visualisierung soll unabhängig von der Simulation der Dynamik sein und erfolgt daher extern, außerhalb des Simulators.

## 3.3 Simulationssystem

Dieses Kapitel beschreibt das zu entwickelnde Simulationssystem. Es soll den generellen Aufbau und die Struktur des Systems wiedergeben. Dabei wird auf die Konzeption und die Schnittstellen zwischen den einzelnen Komponenten eingegangen.

### 3.3.1 Konzeption des Simulationssystems

Gemäß den im vorigen Kapitel gestellten Anforderungen wird ein entsprechendes Konzept für das Simulationssystems entwickelt. Die Abbildung 3-2 stellt den generellen Aufbau dieses Systems grafisch dar. Die gestrichelten Pfeile symbolisieren den logischen und die ausgefüllten Pfeile den tatsächlichen Datenfluss.

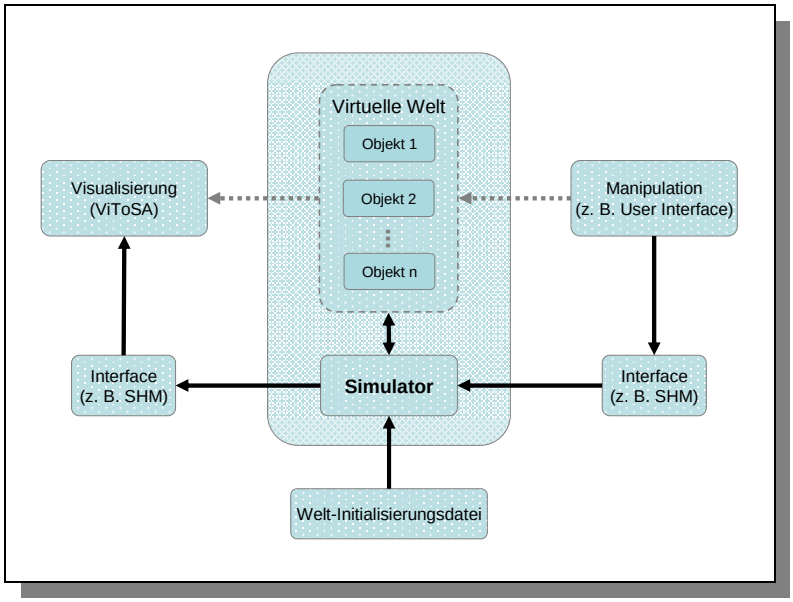



Abbildung 3-2: Konzeption des Simulationssystems, Quelle: Eigene Darstellung.

Das Simulationssystem besteht im Kern aus dem Simulator, mit dem die virtuelle Welt, also die Umgebung des Roboters, erstellt und simuliert wird. Aufbau und Eingriff in die virtuelle Welt erfolgt über die HSimL-API, auf die im Kapitel 3.4 noch näher eingegangen wird.

Das Konzept sieht weiter, vor die Visualisierung der virtuellen Welt außerhalb des Simulators stattfinden zu lassen und so die Simulation unabhängig vom visuellen Aspekt durchzuführen. Wie schon in der Aufgabenstellung erwähnt, wird ViToSA zur Visualisierung eingesetzt. Eine Schnittstelle ermöglicht die Kommunikation zwischen Simulator und Visualisierung.

Den Anforderungen entsprechend wird eine weitere Schnittstelle für die Manipulation der Simulation von außen bereitgestellt. Dies soll einen interaktiven Zugriff auf die virtuelle Welt ermöglichen.

Welche Schnittstellen für Visualisierung und Manipulation verwendet werden und wie die Kommunikation zwischen den einzelnen Komponenten abläuft, wird in dem folgenden Kapitel näher beschrieben.

### 3.3.2 Schnittstellen Design

Für die Interprozess-Kommunikation (IPC) zwischen dem Simulator und der Visualisierung bzw. zwischen dem Simulator und Programmen, welche die virtuelle Welt von außen manipulieren, muss eine geeignete Schnittstelle bereitgestellt werden, damit ein Informationsaustausch zwischen den entsprechenden Prozessen stattfinden kann. Generell stehen mehrere Alternativen zur Verfügung wie Shared Memory (SHM), Sockets, Pipes, Queues, FiFo usw.. SHM bietet dabei die wohl schnellste Übertragung von größeren Datenmengen, wohingegen Sockets die Kommunikation auf mehreren verteilten Rechnern ermöglicht. Aufgrund der Vorteile die SHM und Sockets bieten, wurden beide Varianten parallel implementiert, um somit den Anwendern einen größeren Handlungsspielraum zu geben.

Die benötigten Informationen, die zwischen der Visualisierung und Simulation bzw. zwischen Manipulation und Simulation ausgetauscht werden müssen, sind dafür in zwei speziellen Datenstrukturen festgehalten. Nur über diese Datenstrukturen ist die Interprozess-Kommunikation zwischen den einzelnen Komponenten möglich. Die Datenstrukturen müssen allen beteiligten Kommunikationspartnern bekannt sein.

Im Folgenden wird zunächst auf die beiden Schnittstellen SHM und Sockets näher eingegangen und danach werden die beiden implementierten Datenstrukturen beschrieben.

#### Shared Memory (SHM) Schnittstelle

Shared Memory ist ein vom Betriebssystem verwalteter Speicherbereich, der von mehreren Prozessen gelesen und beschrieben werden kann. Die Prozesse müssen sich aber untereinander abgleichen also synchronisieren, damit nicht mehr als ein Prozess lesend und vor allem schreibend auf ein Speichersegment zugreift.

Zunächst muss ein Prozess den Datenspeicher erzeugen. Dieser wird in Linux im Pfad “/dev/shm/“ abgelegt. Anschließend kann jeder beliebige Prozess den Datenspeicher seinem Adressraum hinzufügen, sofern ihm der Speicher bekannt ist.

Shared Memory ist wohl die schnellste Form der Interprozesskommunikation. Nachteilig ist nur, dass bei diesem Verfahren keine explizite Synchronisation stattfindet und die Kontrolle selbst durch Sperrmechanismen realisiert werden muss. Allerdings stehen auch schon Bibliotheken zur Verfügung die einen Sperrmechanismus implementieren.

Die für die Kommunikation mit dem Simulator eingesetzte SHM-Bibliothek vom Honda Research Institute basiert auf dem POSIX Standard (**P**ortable **O**perating **S**ystem **I**nterface for Uni**X**) und hat auch bereits eine Zugriffskontrolle implementiert.[26]

### Socket Schnittstelle

Ein Socket ist eine Schnittstelle zwischen einem Prozess also einer Applikation und dem vom Betriebssystem verwendetem Netzwerk-Protokoll zur Interprozess- oder Netzwerk-Kommunikation. Der Zugriff erfolgt vergleichsweise wie auf Dateien nur das es sich hier nicht um physikalische Dateien handelt, sondern um Kommunikationskanäle. Die Kommunikation über Sockets in Unix/Linux ist Teil des POSIX-Standards. [27]

Der Vorteil von Sockets gegenüber dem Shared Memory ist die Kommunikation sowohl über das Netzwerk wie auch zwischen Prozessen auf einem Rechner. So kann z.B. die Simulation auf einen leistungsstarken Rechner erfolgen und die Visualisierung auf mehreren verteilten Clients im Netzwerk stattfinden. Es ist aber genauso möglich nur auf einem Rechner die Simulation und Visualisierung durchzuführen.

### Datenstruktur für die Visualisierung

Die Datenstruktur für die Visualisierung **SharedDataVisualisation** definiert alle Informationen die von ViToSA benötigt werden, um eine grafische Ausgabe der Simulation zu ermöglichen. In dem Simulator wird für jedes zu visualisierende Objekt eine Instanz dieser Struktur erzeugt. Diese Daten werden dann am Ende jedes Simulationsschritts dem Visualisierungstool über eine der genannten Schnittstellen, entweder Shared Memory oder Sockets übergeben.
Die Datenstruktur setzt sich aus elementaren Datentypen zusammen. Die Verwendung von Zeigern ist allerdings nicht möglich, da ein Zeiger nur innerhalb eines Programmierblocks gültig ist. Daher sind Zeiger für eine Interprozesskommunikation nicht geeignet. Die Abbildung 3-3 zeigt den Aufbau dieser Datenstruktur.

[26] Vgl. Wolf (2006), S. 353 ff.
[27] Vgl. Brause (2004), S. 259.

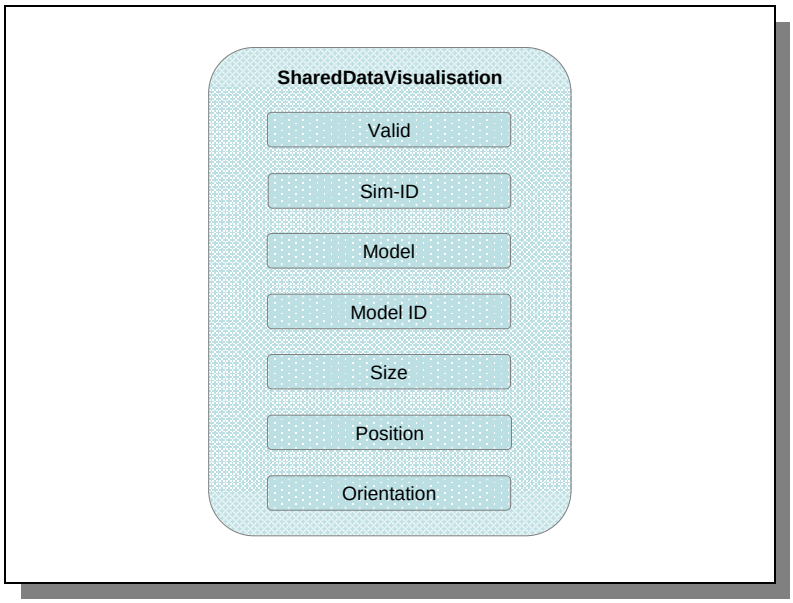



Abbildung 3-3: Datenstruktur SharedDateVisualisation, Quelle: Eigene Darstellung.

**Valid** ist eine spezielle Identifikationsnummer, die jeder richtig erzeugten Instanz der Datenstruktur zugewiesen wird. Die **Sim-ID** ist eine eindeutige Identifikationsnummer, die bestimmt, welcher Simulationswelt dieses Datenobjekt angehört. Der Parameter **Model** legt fest, um was für ein Objektprimitiv es sich handelt, wie z. B. Würfel, Kugel oder Zylinder. Die **Model ID** ist eine eindeutige Identifikationsnummer, die jedem Objekt zugewiesen wird und es genau spezifiziert. Der Parameter **Size** enthält die Größe der Objekte. Bei einem Quader sind das die drei Seitenlängen und bei einer Kugel der Radius. **Position** gibt die genaue Lage des Objekts in Weltkoordinaten an. Dabei ist immer die Position des Massenschwerpunkts gemeint (siehe Kapitel 2.2.1 und 2.5.3). **Orientation** beinhaltet dementsprechend die Orientierung des Objekts.

### Datenstruktur für die Manipulation

Die Datenstruktur SharedDataManipulation ist speziell für die externe Manipulation der virtuellen Welt gedacht. Sie definiert alle Informationen, um ein Objekt in der virtuellen Welt von außen zu steuern und zu modifizieren, wie z. B. die Position des Körpers zu verändern oder einem Körper eine Kraft hinzuzufügen. Die Anwendung, die später für die Manipulation der virtuellen Welt verantwortlich ist, muss lediglich eine Instanz dieser Datenstruktur bilden und die entsprechenden Daten in dem Objekt eintragen. Anschließend übergibt sie das Datenobjekt dem Simulator über eine der beiden Schnittstellen, entweder Shared Memory oder Sockets. Der Simulator selbst prüft vor jedem Simulationsschritt, ob neue Daten für die Manipulation vorliegen und ändert dementsprechend die Simulationseinstellungen. Die Abbildung 3-4 zeigt den Aufbau dieser Datenstruktur.

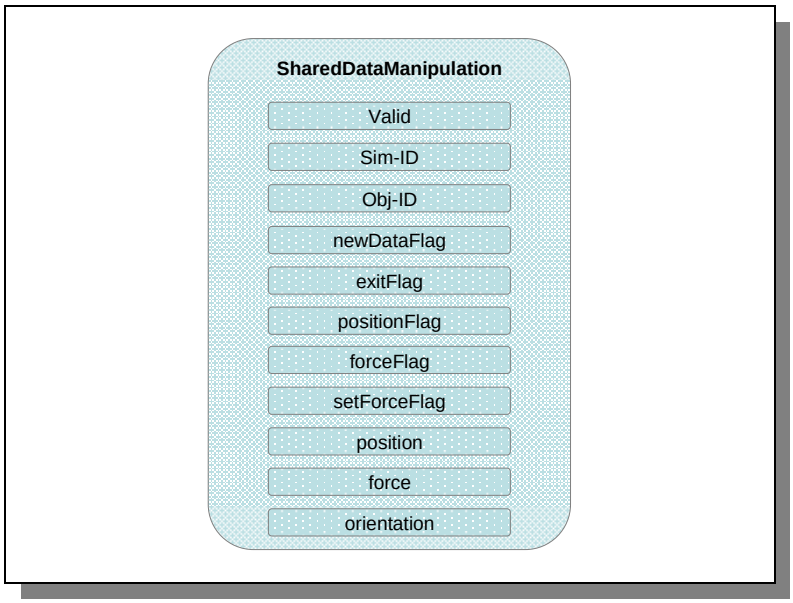



Abbildung 3-4: Datenstruktur SharedDataManipulation, Quelle: Eigene Darstellung.

**Valid** und **Sim-ID** sind entsprechend den Parametern der Datenstruktur für die Visualisierung zu verstehen. **Obj-ID** spezifiziert das zu verändernde Objekt und **newDataFlag** gibt an, ob neue Daten vorliegen. Über das **ExitFlag** ist es möglich die Simulation zu beenden. Die Flags **positionFlag**, **forceFlag** und **setForceFlag** bestimmen, welche Eigenschaften des Körpers geändert werden sollen. Die weiteren Parameter **position** und **force** und **orientation** spezifizieren die neue Position, Kraft oder Orientierung.

### 3.3.3 Ablauf der Kommunikation

Die Abbildung 3-5 zeigt den generellen Ablauf der Kommunikation zwischen Simulator, Visualisierung (ViToSA) und Manipulation. Die drei verschiedenen Anwendungen laufen unabhängig voneinander ab und sind daher getrennt zu betrachten.

Eine externe Anwendung kann über die Schnittstelle Shared Memory oder Sockets in die Simulation eingreifen. Der Simulator liest vor jedem Simulationsschritt die Schnittstelle aus und aktualisiert anhand der eingelesenen Daten die Simulation. Anschließend schreibt er die Simulationsdaten in die Schnittstelle für die Visualisierung. Dieser Vorgang läuft sequentiell ab und wird in einer Schleife ständig wiederholt. Die Visualisierung (ViToSA) liest in bestimmten Abständen den Shared Memory aus und stellt die eingelesenen Daten grafisch dar.

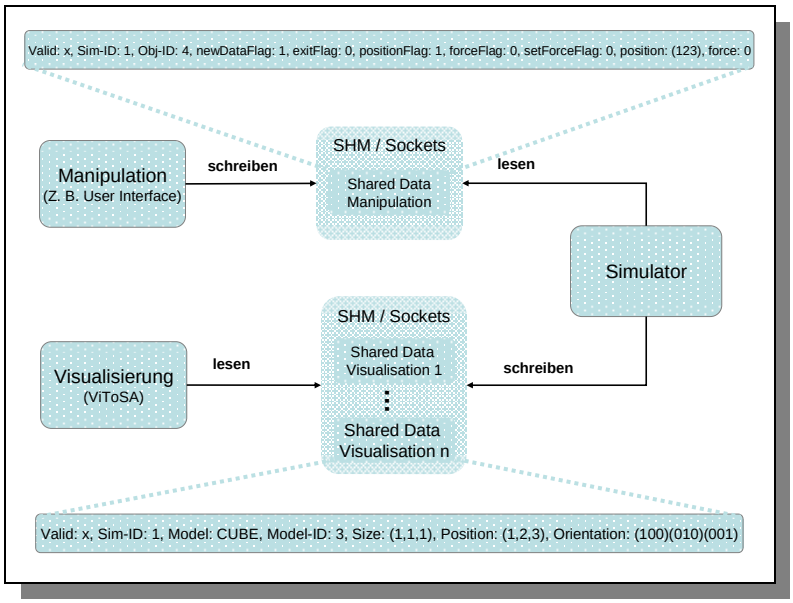



Abbildung 3-5: Ablauf der Kommunikation, Quelle: Eigene Darstellung.

## 3.4 Simulator

Der Fokus der Masterarbeit liegt auf der Simulation der Dynamik starrer Körper. Der hier beschriebene Simulator ist der Kern des gesamten Simulationssystems. Dieses Kapitel gibt einen Überblick über dessen Struktur und Funktionsweise. Im Anhang 2 ist ein Beispielcode für einen Simulator beigefügt.

### 3.4.1 Konzept

Der Aufbau des Simulators wird durch die Abbildung 3-6 widergespiegelt. Basis für den Simulator ist wie schon erwähnt die Open Dynamics Engine. ODE gilt als sehr stabil und ist im Vergleich zu anderen physikalischen Simulationsumgebungen verhältnismäßig schnell und genau. Zudem ist es eine Open-Source Software und somit frei erhältlich. ODE wird dabei für die physikalische Umgebung der virtuellen Welt eingesetzt. Die Einbindung der Physik-Engine in den Simulator erfolgt über die selbst erstellte Simulations-Bibliothek **HSimL**. Diese Bibliothek ist ein Wrapper um ODE und umschließt den Code, um ihn in die unternehmensinterne Systemstruktur zu integrieren. Dadurch wird zusätzlich ein kontrollierter Zugriff auf die ODE Funktionen gewährleistet. Mit Hilfe dieser Bibliothek lassen sich nun virtuelle Welten, bestehend aus starren Körpern und Geometrie-Objekten, erstellen und simulieren. Auch die ODE basierte Kollisionserkennung wird unterstützt. Der Simulator ist mit Hilfe einer gewöhnlichen Textdatei konfigurierbar. Die Textdatei spezifiziert für jeden Körper den Typ (Kugel, Quader, Zylinder usw.), die Größe, die Position und Orientierung. Je nach belieben kann für jeden Körper auch noch ein zusätzliches Geometrie-Objekt angelegt werden. Komplexere

Gebilde sind bisher noch nicht über die Konfigurationsdatei zu erstellen. Dies erfordert zurzeit noch die direkte Programmierung im Simulatorcode. Im Anhang 1 befindet sich eine Beispiel-Konfigurationsdatei. Für die Umsetzung der Textdatei ist ein Parser im Simulator zuständig. Dieser wandelt die in der Datei spezifizierten Objekte in die entsprechenden Körper und Geometrien um. Mit Hilfe der Schnittstellen Shared Memory oder Sockets kann der Simulator nun mit externen Anwendungen, wie z. B. einer HRI-Komponente oder einer einfachen Anwendung kommunizieren.

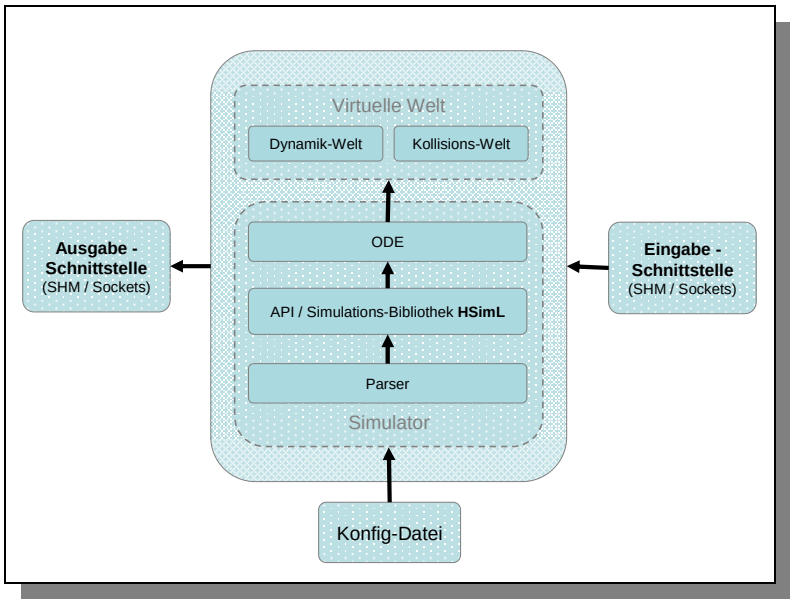



Abbildung 3-6: Architektur des Simulators, Quelle: Eigene Darstellung.

### 3.4.2 Implementierung

Die Implementierung der Simulations-Bibliothek HSimL (**H**RI **Sim**ulation **L**ibrary) erfolgt nach den HRI Richtlinien. Wie bereits an anderer Stelle erwähnt, ist die Simulations-Bibliothek ein Wrapper um die ODE Bibliothek. An dieser Stelle soll nur der generelle Aufbau der API aufgezeigt werden. Eine vollständige Dokumentation aller Funktionen wurde mit Doxygen erstellt und kann aufgrund des großen Umfangs hier nicht angegeben werden. Die API teilt sich in mehrere verschiedene Definitionsdateien auf. Somit hält sich die Struktur weitestgehend an den Aufbau von ODE (siehe Abbildung 3-7).

Die Bibliothek ist aufgeteilt in First-Level API, Second-Level API und Third-Level API. Dabei bilden alle Funktionen aus HWorld und HBody die First- und Second-Level API. Die First-Level API ist für Anwender gedacht, die ohne großen Aufwand eine virtuelle Welt aufbauen möchten. Die Second-Level API dagegen beinhaltet Zusatzfunktionen, wie z. B. das automatische Deaktivieren von Körpern (siehe Kapitel 2.5.3). Diese Funktionen werden nicht unbedingt benötigt, sie sind aber in bestimmten Fällen sehr nützlich. Die restlichen Funktionen

aus HJoint, HSpace, HGeom, HRotation, HContact, HMass bilden die Third-Level API und sind für fortgeschrittene Anwender gedacht, die komplexere Welten erstellen möchten. Hierzu gehört z. B. die Verwendung von Gelenken. Diese Aufteilung ist auch aufgrund des Umfangs der API sehr hilfreich, da es die für den Anwender wichtigen Funktionen zu einer Einheit gruppiert. Funktionen die nicht so häufig oder sehr selten genutzt werden bilden wiederum eine Gruppe für sich.

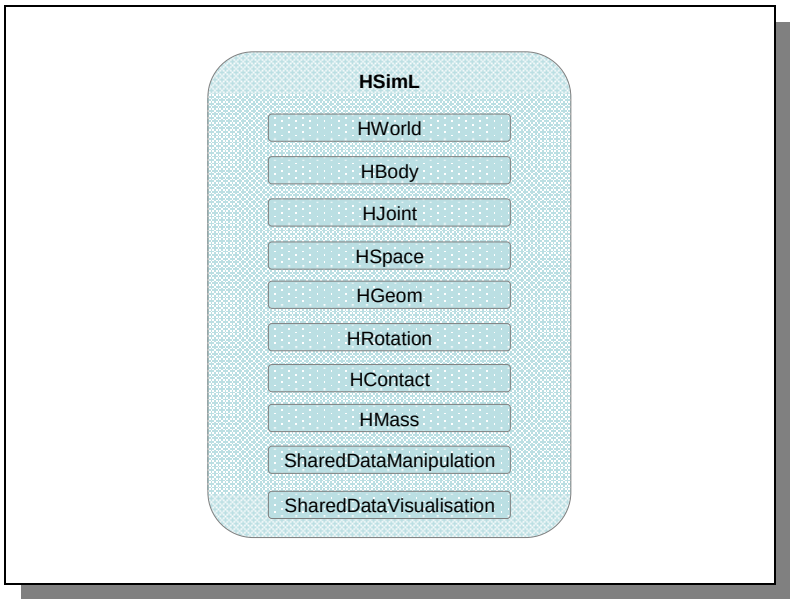



Abbildung 3-7: Simulationsbibliothek HSimL, Quelle: Eigene Darstellung.

1. **HSimL:** Fasst alle Definitionsdateien zu einer Bibliothek zusammen.
2. **HWorld:** Definiert die Funktionen zur Erstellung der Dynamik Welt.
3. **HBody:** Definiert die Funktionen zur Erstellung eines Körpers.
4. **HJoint:** Definiert die Funktionen zur Erstellung von Verbindungen (Gelenken).
5. **HGeom:** Definiert die Funktionen zur Erstellung von Geometrieobjekten.
6. **HSpace:** Definiert die Funktionen zur Erstellung von Kollisionsräumen.
7. **HRotation:** Definiert zusätzliche Funktionen zur Rotation.
8. **HContact:** Definiert Strukturen für die Kontakte.
9. **HMass:** Definiert die Struktur der Masse.
10. **SharedDataManipulation:** Datenstruktur für die Manipulation.
11. **SharedDataVisualisation:** Datenstruktur für die Visualisierung.

Die Abbildung 3-8 zeigt die Struktur der Bibliothek mittels eines UML-Diagramms. Die Welt hat hier eine Kollisionswelt, einen Untergrund und eine Kontaktgruppe bereits integriert. Sie kann aber noch in weitere Kollisionswelten aufgeteilt werden. Den Kollisionswelten wiederum

sind die Kollisionsobjekte zugeordnet. Die Welt besteht weiterhin aus Körpern und Gelenken. Die Körper haben eine eigene Masse und ein Kollisionsobjekt integriert.

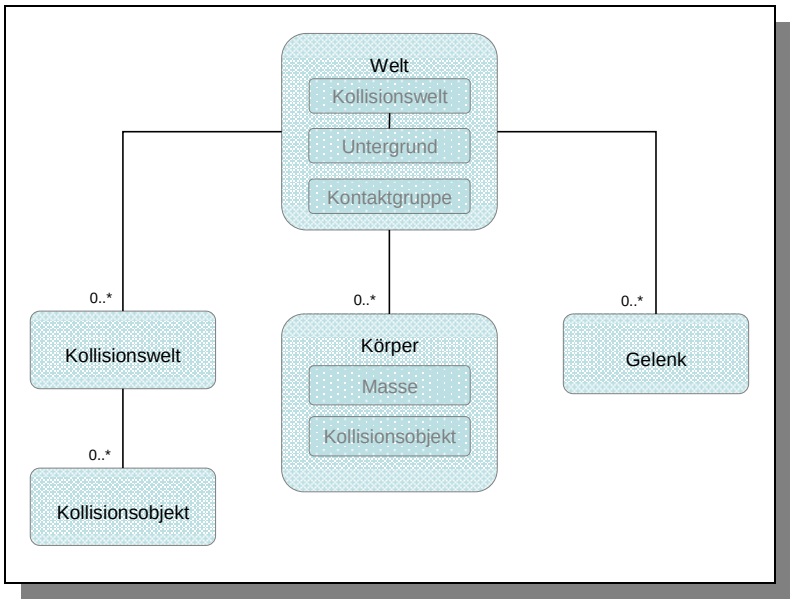



Abbildung 3-8: UML-Diagramm der HSimL-Bibliothek, Quelle: Eigene Darstellung.

### 3.4.3 Simulationsablauf

An dieser Stelle soll nun der Programmablauf des Simulators beschrieben werden. Die Abbildung 3-9 stellt diesen Prozess grafisch dar. Die Erläuterungen beziehen sich dabei auf die Verwendung der SHM-Schnittstelle. Es kann analog dazu auch die Socket-Schnittstelle eingesetzt werden.

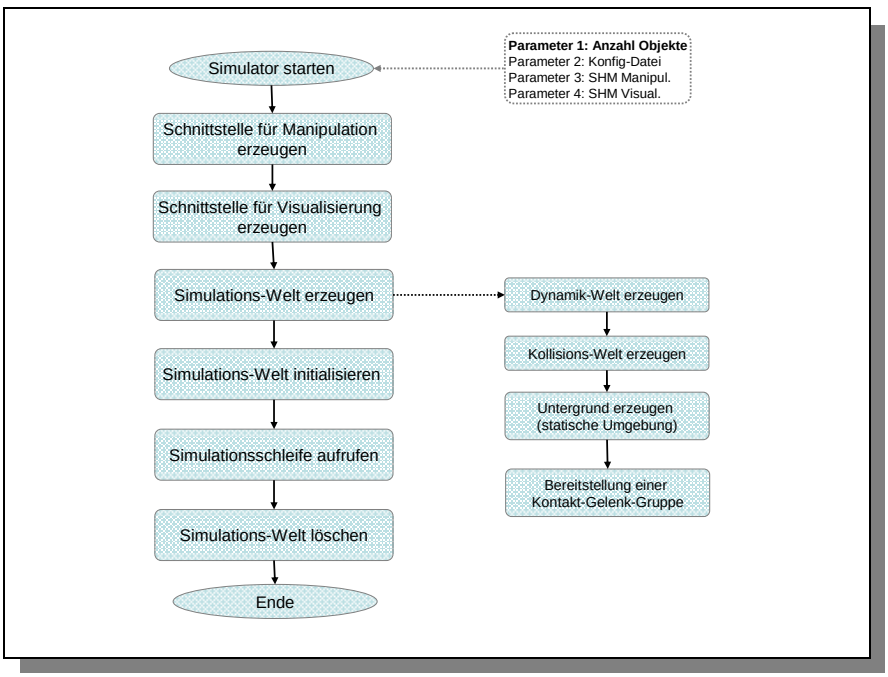



Abbildung 3-9: Programmablaufplan des Simulators, Quelle: Eigene Darstellung.

Dem Simulator muss beim Programmstart die Anzahl der zu simulierenden Objekte bekannt sein. Daher ist die Objektanzahl als Parameter mit zu übergeben. Die übrigen drei Parameter sind optional. Sie sind dafür vorgesehen die Konfigurationsdatei und die Visualisierungs- bzw. die Manipulationsschnittstelle zu spezifizieren. Ohne die Angabe der Parameter wird der Standardname- und Pfad verwendet.

Im nächsten Schritt wird jeweils ein gemeinsamer Speicherbereich (Shared Memory) für die beiden Schnittstellen Visualisierung und Manipulation erzeugt. Die Größe des zu reservierenden Speichers ist dabei abhängig von der Anzahl der Objekte, die simuliert werden sollen. Wird die Socket-Schnittstelle verwendet, müssen an dieser Stelle jeweils ein Netzwerkserver für die Visualisierung und für die Manipulation initialisiert werden.

Anschließend wird die Simulations-Welt erstellt. Dies beinhaltet die automatische Generierung einer Dynamik- und Kollisionswelt, eines Untergrunds (statische Umgebung) und einer Gelenkgruppe für die Kontakt-Verbindungen, die während der Kollisionserkennung generiert werden. Jede virtuelle Welt erhält eine eindeutige ID, mit deren Hilfe die Simulations-Welt eindeutig identifiziert werden kann.

Daraufhin wird die Welt initialisiert. Dies erfolgt über die Konfigurationsdatei, in der alle zu simulierenden Körper genau spezifiziert sind. Der genaue Aufbau dieser Datei ist im Anhang 1 nachzulesen. Abbildung 3-10 macht diesen Prozess anhand eines Ablaufplans deutlich.

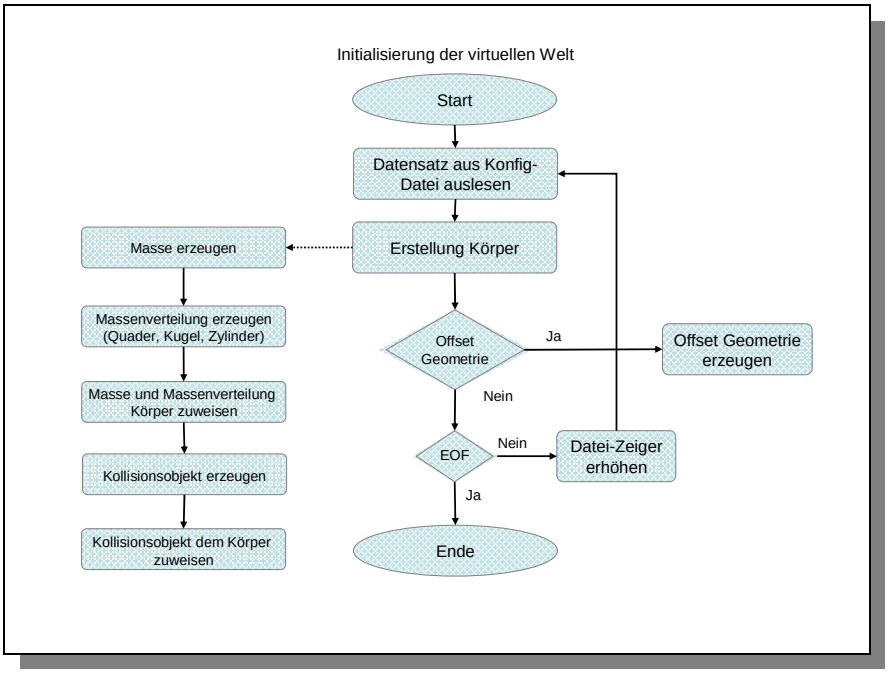



Abbildung 3-10: Programmablauf der Initialisierungsphase, Quelle: Eigene Darstellung.

Die Konfigurationsdatei wird in einer Schleife ausgelesen, bis das Ende der Datei erreicht wird. Für jeden Datensatz wird ein entsprechender Körper konstruiert. Dabei werden zunächst die Masse und die Massenverteilung entsprechend den Angaben in der Konfigurationsdatei erstellt. Danach wird automatisch ein entsprechendes Kollisionsobjekt generiert und dem Körper zugewiesen. Im nächsten Schritt wird überprüft, ob ein Offset-Geometrie-Objekt erzeugt werden soll. Eine Offset-Geometrie ist ein zusätzliches Geometrie-Objekt, das einem Körper zugewiesen werden kann. Durch Angabe eines Offsets kann die Geometrie beliebig am Körper positioniert werden. Damit ist es möglich auch komplexere Objekte zu entwerfen.

Nach der Initialisierung wird die Simulationsschleife aufgerufen, die in der Abbildung 3-11 dargestellt ist. Die Simulationsschleife wird solange durchlaufen bis von außerhalb das Exit-Flag gesetzt wird. In der Simulationsschleife selbst wird am Anfang der Shared Memory der Manipulations-Schnittstelle ausgelesen und eventuell die Simulationsdaten geändert, falls die entsprechenden Flags gesetzt sind. Danach startet der Simulator die Kollisionserkennung und ruft für jedes eventuell kollidierende Objektpaar eine Callback-Funktion (siehe Kapitel Siehe 2.5.11) auf.

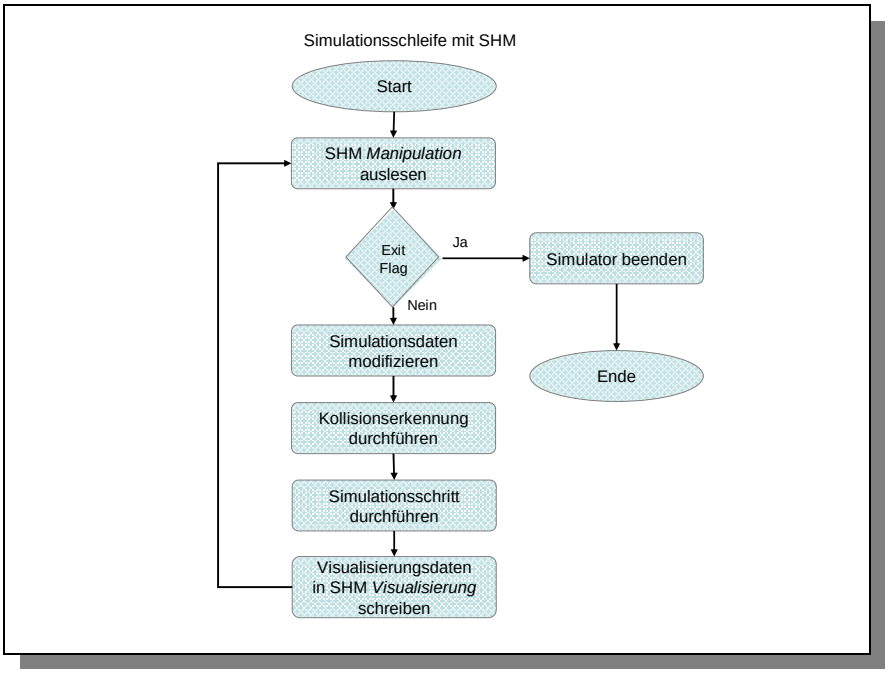



Abbildung 3-11: Programmablauf der Simulationsschleife, Quelle: Eigene Darstellung.

Nach Beendigung der Kollisionserkennung wird der eigentliche Simulationsschritt ausgeführt und anschließend die symbolischen Informationen an die Visualisierung übergeben, indem sie in den Shared Memory der Visualisierungs-Schnittstelle geschrieben werden.

### 3.4.4 Testszenarios

Die hier beschriebenen Testszenarios haben den Zweck, die implementierte API zu testen. Dabei wird überprüft, ob die Funktionen in der Simulation das richtige Verhalten aufzeigen und die richtigen Rückgabewerte liefern.

Die Verifizierung der API (HSimL-Bibliothek) erfolgt durch die Erstellung von Testprogrammen, die sich an die Struktur der ODE-Simulationsprogramme anlehnen (siehe Kapitel 2.5.13). Auch die Visualisierung der Tests erfolgt der Einfachheit halber mit der in ODE enthaltenen Grafik-Bibliothek Drawstuff. Erst in einem späteren Schritt wird für die Visualisierung das Grafikprogramm von HRI eingesetzt.

In den generierten Testszenarios sollen die einzelnen Funktionen der API auf ihre richtige Funktionsfähigkeit überprüft werden. Dafür werden aufgrund des großen Umfangs der API zunächst nur die wesentlichen Funktionen getestet. Dazu gehört z. B. das erzeugen von Körpern, Kollisionsobjekten und Gelenken zur Verbindung von Körpern oder das Hinzufügen von Kräften auf einen Körper und viele weitere mehr.

## 3.5 Manipulation

Die Simulation von starren Körpern soll von außen beeinflussbar sein. Dadurch ist es möglich, während einer aktiven Simulation Modifikationen an einzelnen Objekten vorzunehmen. Die Simulation an sich darf jedoch nicht beeinflusst werden oder sogar aussetzen, sondern muss unabhängig von potentiellen Änderungen weiterlaufen. Es soll schließlich gewährleistet werden, dass die Simulation die Realität so gut wie möglich nachbildet.

Generell ist die Manipulation der virtuellen Welt auf zwei verschiedene Arten möglich. Zum einen direkt im Simulator selbst durch die Anwendung der API-Funktionalität und zum anderen durch die interaktive Manipulation mit Hilfe einer geeigneten Schnittstelle. Diese interaktive Manipulation soll im Folgenden beschrieben werden.

Die Kommunikation zwischen der externen Anwendung und dem Simulator erfolgt wie schon im Kapitel 3.3.2 beschrieben entweder durch die Verwendung des Shared Memory oder über Sockets. In diesem Fall soll die Kommunikation anhand der SHM-Schnittstelle beschrieben werden. Dabei wird der Informationsaustausch mittels einer speziell dafür entworfenen Datenstruktur **SharedDataManipulation** realisiert (siehe Kapitel 3.3.4). Diese Datenstruktur definiert alle nötigen Informationen, die für die Manipulation eines Objekts erforderlich sind. Die Abbildung 3-5 im Kapitel 3.3.3 macht den Prozess der interaktiven Manipulation grafisch deutlich.

Bei dem interaktiven Eingriff in die Simulation über den Shared Memory wird für den zu manipulierenden Körper eine Instanz dieser Datenstruktur mit den entsprechenden Soll-Objektwerten erzeugt und an den Simulator über die Schnittstelle übergeben. Der Simulator liest die Schnittstelle aus und führt die Änderungen durch.

Die folgende Abbildung 3-12 macht den Einleseprozess des Simulators aus der SHM-Schnittstelle anhand eines Programmablaufplans deutlich.

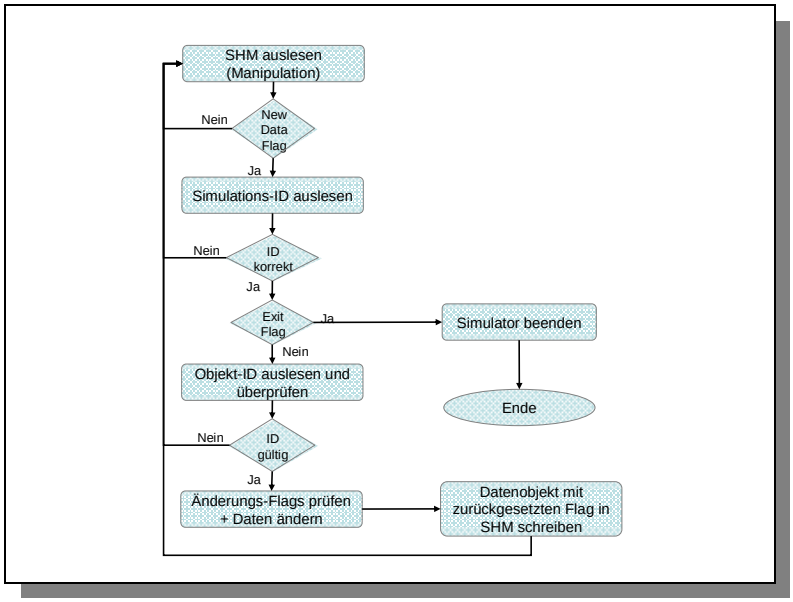



Abbildung 3-12: Programmablauf des Einleseprozesses, Quelle: Eigene Darstellung.

Nach dem Einlesen der Daten wird zunächst geprüft, ob das *newDataFlag* gesetzt ist. Wenn das der Fall ist, weiß der Simulator, dass sich die Daten geändert haben und liest die Simulations-ID aus. Wenn die ID mit der ID der Simulationswelt übereinstimmt, wird der Einleseprozess weitergeführt und überprüft, ob das *exitFlag* gesetzt ist. Ist auch das der Fall, wird der Simulator dementsprechend beendet. Ansonsten ist zu überprüfen, dass die spezifizierte Objekt-ID auch wirklich einen existierenden Körper in der Simulation zugeordnet werden kann. Wenn auch das zutrifft, werden die Daten entsprechend den Änderungs-Flags mit den angegebenen Daten modifiziert.

## 3.6 Visualisierung

Die Visualisierung der Simulation erfolgt mit dem im Kapitel 2.6 beschriebenen Programm ViToSA. Auch hier wird die Kommunikation zwischen dem Visualisierungsprogramm und dem Simulator anhand der SHM-Schnittstelle beschrieben. Die Kommunikation ist aber auch über die Socket-Schnittstelle möglich.

Der Informationsaustausch erfolgt mittels einer speziell dafür entworfenen Datenstruktur **SharedDataVisualisation** (siehe Kapitel 3.3.4) ähnlich wie bei der Manipulation. Diese Datenstruktur definiert alle nötigen Informationen, die für die Visualisierung eines Objekts erforderlich sind. Über diese Struktur findet auch die angeforderte Statusabfrage von Körpern in der virtuellen Welt statt, wie z. B. die aktuelle Position oder Geschwindigkeit eines Objekts. Die Abbildung 3-5 im Kapitel 3.3.3 macht den Prozess der Kommunikation grafisch deutlich.

Der Ablauf der Visualisierung kann nun wie folgt beschrieben werden. Für jeden an der Simulation teilnehmenden Körper wird eine Instanz, also ein Datenobjekt der Datenstruktur SharedDataVisualisation erzeugt und in einem Daten-Array gespeichert. Nach jedem Simulationsschritt werden vom Simulator die symbolischen Informationen, also Position, Orientierung usw., aller beteiligten Körper in dem zugehörigen Datenobjekt festgehalten. Anschließend wird das gesamte Daten-Array in den Shared Memory geschrieben. Das Visualisierungsprogramm liest nun in einem bestimmten Intervall das Daten-Array aus und generiert aus den symbolischen Informationen visuelle Daten.

Für die Visualisierung der virtuellen Welt muss ViToSA zunächst durch einen Konverter erweitert werden. Es sind jeweils zwei verschiedene Konverter für die Kommunikation über Shared Memory oder über Sockets zu implementieren, wie es die Abbildung 3-13 illustriert.

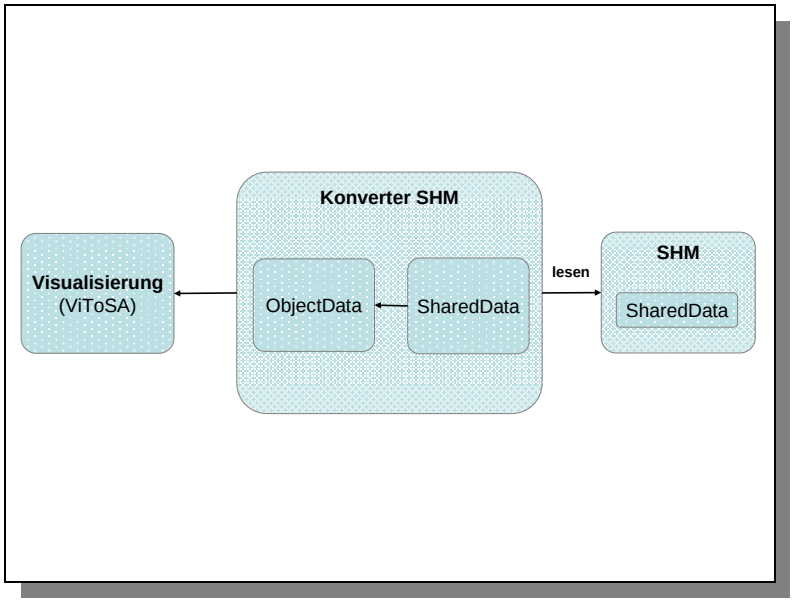



Abbildung 3-13: Kommunikationsablauf zwischen SHM und ViToSA, Quelle: Eigene Darstellung.

Der Konverter liest dabei die Daten über Shared Memory oder Socket ein, konvertiert die Daten in das entsprechende Format ObjectData und übergibt die Informationen an ViToSA. Die Abbildungen 3-14 zeigt ein Beispiel für die Visualisierung einer virtuellen Welt.

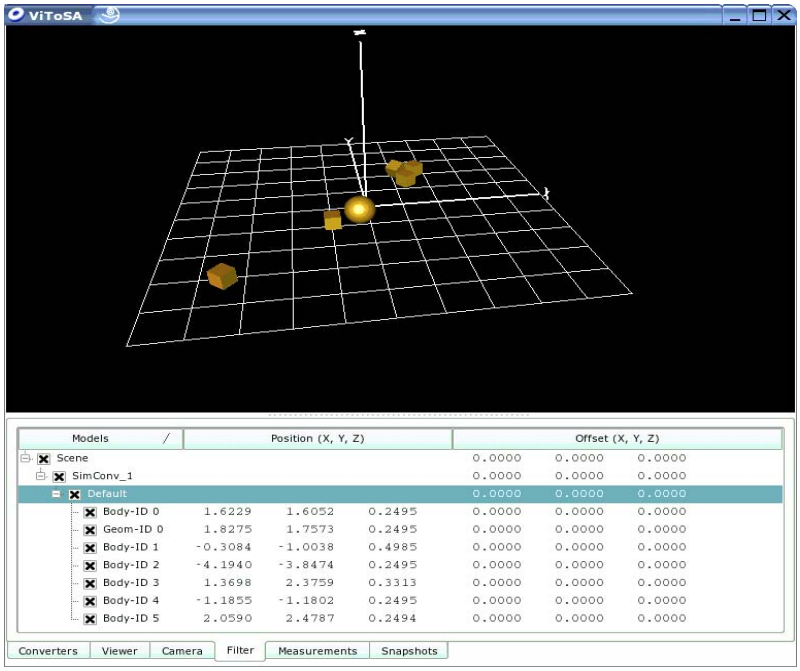



Abbildung 3-14: Screenshot einer Beispiel-Simulationsumgebung mit ViToSA, Quelle: HRI.

## 3.7 Testszenarios

Das Simulationssystem soll, wie schon in der Aufgabenstellung beschrieben, durch einen geschlossenen Regelkreis (Closed-Loop) mit Hilfe des bei HRI verwendeten Komponenten-/Integrationsmodell getestet werden. Das Testszenario umfasst allerdings nur den einfachen Regelkreis aus der Aufgabenstellung (siehe Kapitel 1.3, sowie Abbildung 1-1). Der zweite Regelkreis wurde nicht realisiert.

Die Abbildung 3-15 zeigt den Aufbau des Testszenarios. Im oberen Bereich sind die beiden Komponenten **Robot-Control** und **Manipulate-Object** abgebildet, die zusammen den Regelkreis bilden. Die Robot-Control Komponente hat in dieser Darstellung nur eine symbolische Bedeutung. In Realität ist sie ein System von Komponenten, die den Roboter simulieren. Diese Komponente wird von HRI zur Verfügung gestellt. Im unteren Bereich ist der Simulator dargestellt. Er wird, wie schon erwähnt, als Stand-Alone-Server betrieben. Der Ablauf kann nun wie folgt beschrieben werden:

Der Roboter, der durch die Komponente Robot-Control simuliert wird, bewegt seine Hand in die Nähe eines Objekts und greift danach. Das zu greifende Objekt wird bei der Berührung mit der Roboterhand deaktiviert und somit von der Simulation ausgeschlossen. Das Objekt wird jetzt von außen, also von der Robot-Control Komponente, gesteuert. Die Steuerung erfolgt, indem die Komponente Robot-Control die Koordinaten der Hand an die Komponente Manipulate-Object schickt. Die Position der Hand wird an den Simulator übergeben. Die

Kommunikation erfolgt dabei über Shared Memory. Die neue Ist-Position wird der Komponente vom Simulator zurückgegeben. Diese wird sogleich an die Komponente Robot Control übertragen.

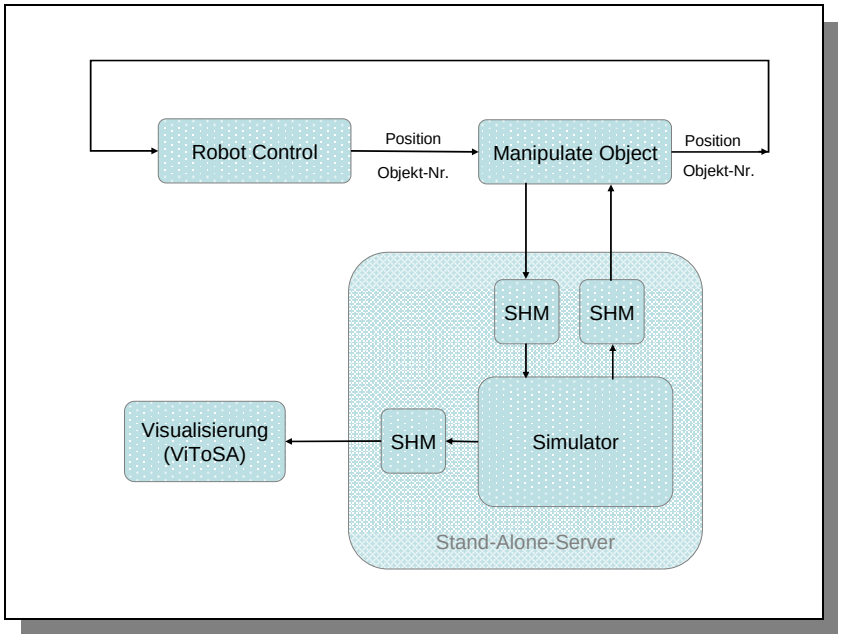



Abbildung 3-15:Darstellung des Test-Regelkreises (Closed-Loop), Quelle: Eigene Darstellung.

# 4 Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurde ein Simulationssystem konzipiert und entwickelt, das die Dynamik starrer Körper simuliert, visualisiert und aktive Simulationen von außen manipulieren kann. Das System integriert außerdem eine Kollisionserkennung.

Die in der Aufgabenstellung und Zielsetzung genannten Anforderungen an das zu entwickelnde System wurden alle umgesetzt. Aktive Simulationen können nun über eine Schnittstelle von außen manipuliert werden. Die Manipulation der virtuellen Welt kann über eine externe Anwendung erfolgen, die über die beiden Schnittstellen Shared Memory oder Sockets auf die virtuelle Welt zugreift. Auch die Visualisierung der virtuellen Welt mit dem HRI internen Grafikprogramm ViToSA, ist problemlos möglich. Die Simulations-API ist den Anforderungen entsprechend umgesetzt worden. Mit ihr können nun virtuelle Welten aufgebaut und die Umgebung des Roboters nachgebildet werden.

Im Laufe des Projekts mussten aufgrund veränderter Bedingungen allerdings Änderungen bei der Umsetzung des Simulationssystems durchgeführt werden. Hier ist vor allem der Simulator anzuführen, der jetzt als eine auszuführende Einheit, also als ein Programm realisiert wurde und nicht wie am Anfang vorgesehen als eine Komponente des HRI-Modells. Es hat sich erst im Laufe des Projekts herausgestellt, dass ODE nicht threadsicher ist und somit der einwandfreie Einsatz im threadbasierten Betriebssystem des Unternehmens nicht garantiert werden konnte. Daraus ergaben sich wiederum weitere Änderungen an das gesamte Simulationssystem. So konnte der Zugriff auf den Simulator nicht mehr direkt durch die Simulations-API erfolgen, sondern musste über entsprechende Schnittstellen (Shared Memory) realisiert werden. Anstatt der Simulator-Komponente wurde eine Ersatzkomponente implementiert, die als Input die Soll-Position eines Körpers erhält und als Output die Ist-Position des Objektes wieder ausgibt.

Die übrigen Anforderungen an den Simulator wurden entsprechend umgesetzt. Virtuelle Welten, in der die Dynamik starrer Körper simuliert wird, sind auf einfache Art und Weise zu erstellen. Vor allem durch die Konzeption eines Textparsers können nun mittels einer normalen Textdatei virtuelle Welten auf einfache Weise aufgebaut werden. Eine Bespielkonfiguration ist im Anhang 1 beigefügt. Allerdings können über diese Textdatei noch keine Verbindungen (Gelenke) zwischen den Körpern erstellt werden. Diese Funktionalität wäre in einer späteren Version zu realisieren. Es sei hier noch angemerkt, dass der Textparser nicht Teil der Anforderungen war, sondern aufgrund der veränderten Bedingungen erst im Laufe des Projekts noch zusätzlich spezifiziert wurde.

Die Simulation von virtuellen Welten unter Verwendung der Socket-Schnittstelle hat bisher noch den Nachteil, dass bei der Initialisierung des Simulators solange gewartet werden muss, bis sich die beiden Clients für die Visualisierung und Manipulation mit dem Netzwerkserver verbinden. Dieses Problem ist durch einen Multithread-Ansatz zu lösen, indem der Simulator in mehrere Threads aufgeteilt wird. Der Hauptthread simuliert die virtuelle Welt, während nebenher ein anderer Thread auf die Clients (Visualisierung und Manipulation) wartet und sobald sich ein Client anmeldet die Verbindung zu ihm erstellt. Die Multithread-Erweiterung konnte leider nicht mehr im Zeitrahmen der Masterarbeit realisiert werden und ist daher in einer späteren Version zu ergänzen. Es sei hier noch angemerkt, dass diese Erweiterung allerdings nicht zu den Anforderungen gehörte.

Ursprünglich war es beabsichtigt, sich nicht nur auf eine Simulationsbibliothek zu beschränken, sondern es sollte die Möglichkeit geben mehrere Bibliotheken zu benutzen. Dies würde zum einen eine gewisse Unabhängigkeit gewährleisten aber zum anderen auch die Möglichkeit bieten verschiedene mehr oder weniger komplexe Implementierungen einer Funktion zu nutzen. Ferner sollte z. B. auch eine eigene einfach gehaltene Bibliothek für die Simulation der Dynamik entwickelt werden, um komplexe Funktionen z. B. aus ODE durch einfachere, weniger rechenintensive Funktionen zu ersetzen. Um diese Anforderung zu ermöglichen wurde ein entsprechendes Konzept für ein Framework entwickelt. Dieses Framework sollte eine API bereitstellen, die wiederum verschiedene Funktionen mit unterschiedlicher Implementierung aus mehreren Bibliotheken unterstützt. Eine intensive Erprobung der ODE-Bibliothek machte aber immer mehr deutlich, dass die frei erhältliche Physik-Engine allen gestellten Anforderungen an den Simulator bereits entspricht und somit eine Eigenentwicklung keinen höheren Nutzen hervorbringen würde. Daraufhin wurde auch auf die Entwicklung des Frameworks verzichtet und ODE als alleinige Simulations-Bibliothek eingesetzt.

Die Open Dynamics Engine als Basis des Simulators hat sich in den Testdurchläufen und während der Erprobung des Simulators als sehr stabil und wenig fehleranfällig hervorgehoben, solange die Simulationsschritte klein gehalten wurden $(\Delta t < 0.005)$. Durch ODE bekommt der Anwender ein mächtiges Werkzeug für die Konstruktion und einfache Handhabung komplexer Systeme in die Hand gelegt. Durch den ERP und CFM Parameter können auf einfache Weise Feinabstimmungen in der virtuellen Welt gemacht werden.

Wie bereits erwähnt, hat sich erst gegen Ende des Projekts rausgestellt, dass ODE nicht threadsicher ist, d. h., dass eine fehlerfreie Ausführung von mehreren parallelen Simulatoren über Threads nicht garantiert werden kann. Nur die Ausführung von parallelen Simulatoren über getrennte Prozesse ist möglich. Dies ist insofern ein Problem, da das beim HRI einge-

setzte Betriebssystem RTBOS (**R**eal **T**ime **B**rain **O**perating **S**ystem) auf die Ausführung von Threads ausgelegt ist. Eine Möglichkeit dieses Problem zu lösen, wäre die HSimL-Bibliothek dazu benutzen, um ODE threadsicher zu machen. Dafür müssten alle Daten, auf die ein globaler Zugriff möglich ist, durch einen bestimmten Sperrmechanismus geschützt werden. Aber alleine das Ausfindig machen der entsprechenden Daten ist sehr schwer und zeitaufwendig, zumal der gesamte Prozess für jede neue Version von ODE wiederholt werden müsste. Daher wurde darauf verzichtet und der Simulator jetzt nur noch als ein auszuführendes Programm implementiert.

Ein Ziel, das mit der Erstellung der eigenen Simulations-Bibliothek HSimL verfolgt wurde, war es, den eigentlichen ODE-Code vor dem Anwender zu verbergen. Bei der Einbindung der Kollisionserkennung gab es hier aber erhebliche Probleme. Diese erfolgt bei ODE nämlich über eine Callback-Funktion, die erst im Anwendungsprogramm (Simulator) spezifiziert und als Funktionszeiger übergeben wird. Diese Funktion erwartet ODE spezifische Parameter, die nicht durch ein Casting umzuwandeln sind. Daher wurde hier zunächst direkt auf den ODE-Code zurückgegriffen. Gewiss wäre es möglich, den frei zugänglichen ODE Code zu modifizieren und somit auch diese Callback-Funktion abzuändern. Nachteilig wäre aber, dass für jedes neue Release von ODE diese Änderungen wiederholt werden müssten. Das Problem wurde im Endeffekt gelöst, indem die Kollisionserkennung einfach fest in die Bibliothek implementiert wurde. Dadurch kann die Kollisionserkennung zwar nicht mehr ohne eine Neuübersetzung der Bibliothek konfiguriert werden, aber der Anwender braucht sich dafür auch nicht mehr um die Kollisionserkennung zu kümmern.

Die Visualisierung und Manipulation sind wie gefordert unabhängig von der eigentlichen Simulation mittels eines geeigneten Interface realisiert. Tatsächlich wurden zwei verschiedene Schnittstellen für die Kommunikation zwischen den einzelnen Komponenten ViToSA, Manipulation und Simulator implementiert. Der Informationsaustausch kann entweder über Shared Memory oder aber mit Hilfe von Sockets erfolgen. Die Umsetzung von zwei unterschiedlichen Modellen war zwar nicht Teil der Anforderungen, aber durch die Kombination der beiden Varianten ergaben sich gewisse Vorteile. Zum einen kann die Simulation, Visualisierung und Manipulation jetzt von verteilten Rechnern im Netzwerk stattfinden, zum anderen überragt die Shared Memory Variante durch ihre Schnelligkeit. Durch die Umsetzung beider Konzepte bekommt der Anwender auch einen größeren Handlungsspielraum. Für die Implementierung des Shared Memory wurde zunächst die unternehmenseigene IOStream-Bibliothek verwendet. Aufgrund eines neuen Release dieser Bibliothek, bei der keine Zugriffssteuerung mehr wegen der Gefahr von Deadlocks unterstützt wird, musste auf die jetzige SHM-

Bibliothek zurückgegriffen werden, die auf den POSIX-Standard aufbaut. Die Implementierung der Socket-Schnittstelle erfolgte mit der unternehmenseigenen Socket-Bibliothek.

Die Entwicklung und Verwendung der beiden Datenstrukturen SharedDataVisualisation und SharedDataManipulation für die Visualisierung bzw. für die Manipulation erweitern das Konzept der Unabhängigkeit und der Erweiterbarkeit des Simulationssystems, da die beiden Datenkonzepte jederzeit mit neuen zusätzlichen Informationen erweitert werden können.

Die Visualisierung der Simulation konnte problemlos mit Hilfe eines Konverters in ViToSA verwirklicht werden. Es steht jeweils ein Konverter für das Shared Memory Modell und ein Konverter für das Netzwerk Modell zur Verfügung. Ein kleiner Nachteil der Visualisierung mit ViToSA ist, dass es versucht alle Körper im Sichtfeld des Benutzers zu behalten. Die Folge davon ist, dass ViToSA den Sichtbereich immer weiter vergrößert, falls sich ein Körper z. B. von Zentrum weg bewegt.

Für die interaktive Manipulation der Simulation wurde eine kleine Anwendung (Benutzerschnittstelle) implementiert, über die eine textbasierte Eingabe mittels der Tastatur die virtuelle Welt manipuliert werden kann. In einem weiteren Schritt wäre die Benutzerschnittstelle um die Interaktion mit einer (Space-) Mouse noch zu erweitern.

Die Entwicklung und Dokumentation des Projekts hat aus Zeitgründen parallel stattgefunden.

# Literaturverzeichnis

**Brause, R. (2004):** Betriebssysteme – Grundlagen und Konzepte, 3. überarbeitete Auflage, Berlin (2004).

**Baraff, D. (1997):** An Introduction to Physically Based Modeling: Rigid Body Simulation I – Unconstrained Rigid Body Dynamics, online verfügbar unter www.cs.edu/ ~baraff/pbm/pbm.html (2006).

**Dreizler, R.M. / Lüdde, C.S. (2003):** Theoretische Physik 1 – Theoretische Mechanik, Berlin/Heidelberg (2003).

**Eckstein, J. (1998):** Echtzeitfähige Kollisionserkennung für Virtual Reality Anwendungen. Dissertation, Technische Fakultät der Universität des Saarlandes (1998).

**Gross, D. / Hauger, W. / Schnell, W. (1998):** Technische Mechanik 1 – Statik, 6. Auflage, Berlin (1998).

**Honda Research Institute Europe GmbH (2006):** http://www.honda-ri.de/.

**Honerkamp, J. / Römer, H. (1989):** Klassische Theoretische Physik – Eine Einführung, 2. Auflage, Berlin/Heidelberg (1989).

**Mühlig, M. (2006):** VITOSA – A 3D Visualisation Tool for Simulation Applications, HRI internal Report (2006).

**Myers Enzyklopädisches Lexikon in 25 Bänden (1977):** Band 19, 9. Auflage, Mannheim (1977).

**Myers Enzyklopädisches Lexikon in 25 Bänden (1977):** Band 21, 9. Auflage, Mannheim (1977).

**Open Dynamics Engine (2006):** http://www.ode.org/.

**Orear, J. (1982):** Physik, Band 1, München (1982).

**Smith, R. (2006):** http://www.q12.org.

**Warken, T. (2004):** Collision Detection for Curved Rigid Objects in the Context of Dynamics Simulations. Dissertation, Naturwissenschaftlich-Technische Fakultät der Universität des Saarlandes (2004).

**Wolf, J. (2006):** Linux-Unix-Programmierung – Das umfassende Handbuch, 2. erweiterte Auflage, Bonn (2006).

# Anhang

### Anhang 1: Beispiel für eine Konfigurationsdatei - “KonfigSimulation.txt“

Im Folgenden wird eine Beispielkonfiguration für die Initialisierung des Simulators aufgelistet. Ein Datensatz wird dabei immer in die beiden Bezeichner OBJEKT_START und OBJEKT_END eingeschlossen, um die Zusammengehörigkeit der eingeschlossenen Daten zu einem Objekt zu kennzeichnen. Nach dem Start-Bezeichner wird der Objekttyp angegeben. Danach folgt die Position, Größe, Kraftvektor, Drehmomentvektor und das Flag *AddGeom*. Wird dieses Flag gesetzt, kann dem Körper noch ein Kollisionsobjekt hinzugefügt werden, wodurch sich auch etwas komplexere Objekte erstellen lassen. Wenn das Flag gesetzt ist, muss Typ, Offset-Position und Größe des zusätzlichen Kollisionsobjekts angegeben werden.

```
OBJEKT_START
Model= CUBE
Position= 1 1 1
Size= 0.5 0.5 0.5
Force= 0 0 0
Torque= 0 0 0
AddGeom= 0
OBJEKT_END

OBJEKT_START
Model= CUBE
Position= 2 2 2
Size= 0.5 0.5 0.5
Force= 0 0 0
Torque= 0 0 0
AddGeom= 1
GeomOffsetType= CUBE
GeomOffsetPos= 0.2 0 0
GeomOffsetSize= 0.2 0.2 0.2
OBJEKT_END

OBJEKT_START
Model= CYLINDER
Position= -1 -1 1
Size= 0.5 0 0
Force= 0 0 0
Torque= 0 0 0
AddGeom= 0
OBJEKT_END

OBJEKT_START
Model= SPHERE
Position= -1 -1 1
Size= 0.5 0 0
Force= 0 0 0
AddGeom= 0
OBJEKT_END
```

### Anhang 2: Simulator-Code

Der nachfolgende Code zeigt ein Beispiel für einen ganz einfachen Simulator. Dafür wird eine Welt und ein Körper erstellt. Die Welt hat bei der Initialisierung standardmäßig eine Gravitationskraft von (0, 0, -9.81). Als Körper wird ein Quader mit einer Seitenlänge (1, 1, 1), der Position (0, 0, 0) und der Masse 1 erzeugt. Als Motion-mode wird hier eine 0 übergeben, d. h. der Körper wird im dynamischen Modus gestartet, anstatt im Kinematik-Modus. Anschließend wird die Simulationsschleife aufgerufen. Hier wird zu dem Körper in jedem Durchgang eine Kraft addiert und danach ein Simulationsschritt durchgeführt. Anschließend erfolgt die Visualisierung der virtuellen Welt. Nach beenden der Simulationsschleife, ist die Welt noch zu bereinigen.

```
#include<HSimL.h>

int main(int argc, char* argv[])
{
    HWorld *simWorld;
    HBody *body;

    double size[3] = { 1, 1, 1 };
    double position[3] = { 0, 0, 0 };
    double force[3] = { 0.2, 0, 0 };

    // create virtual world
    simWorld = HWorld_new( );
    HWorld_init( simWorld );

    // create and init body
    body = HBody_new( );

    // HBody_init( "Body", "World", "Mass", "Type", "Size", "Position", "Motion-mode");
    HBody_init( body, simWorld, 1, CUBE, size, position, 0);

    // Simulation loop
    while( 1 )
    {
        HBody_addForce( body, force );
        HWorld_step( simWorld );
        visualisation();
    }

    // cleaning the world
    HBody_clear( body );
    HBody_delete( body );

    HWorld_clear( simWorld );
    HWorld_delete( simWorld );

    return 0;
}
```